Concepts of Programming with Java

Sébastien Jodogne, Ramin Sadre, Pierre Schaus

Jan 10, 2025

CONTENTS

CHAPTER

ONE

FROM PYTHON TO JAVA

Chapter 1 of this book is intended for students and hobbyists who are already familiar with the basics of Python programming, i.e., they know how to use variables, lists, functions, and plain data objects. A deeper knowledge of object-oriented programming is not required.

The goal of the following sections is to make you quickly familiar with the important differences between Python and Java and with the basic object oriented mechanisms of Java. More advanced topics, such as interfaces, abstract classes, or lambda functions, will be seen in the subsequent parts of the book.

1.1 Your first Java program with IntelliJ

1.1.1 Installing IntelliJ

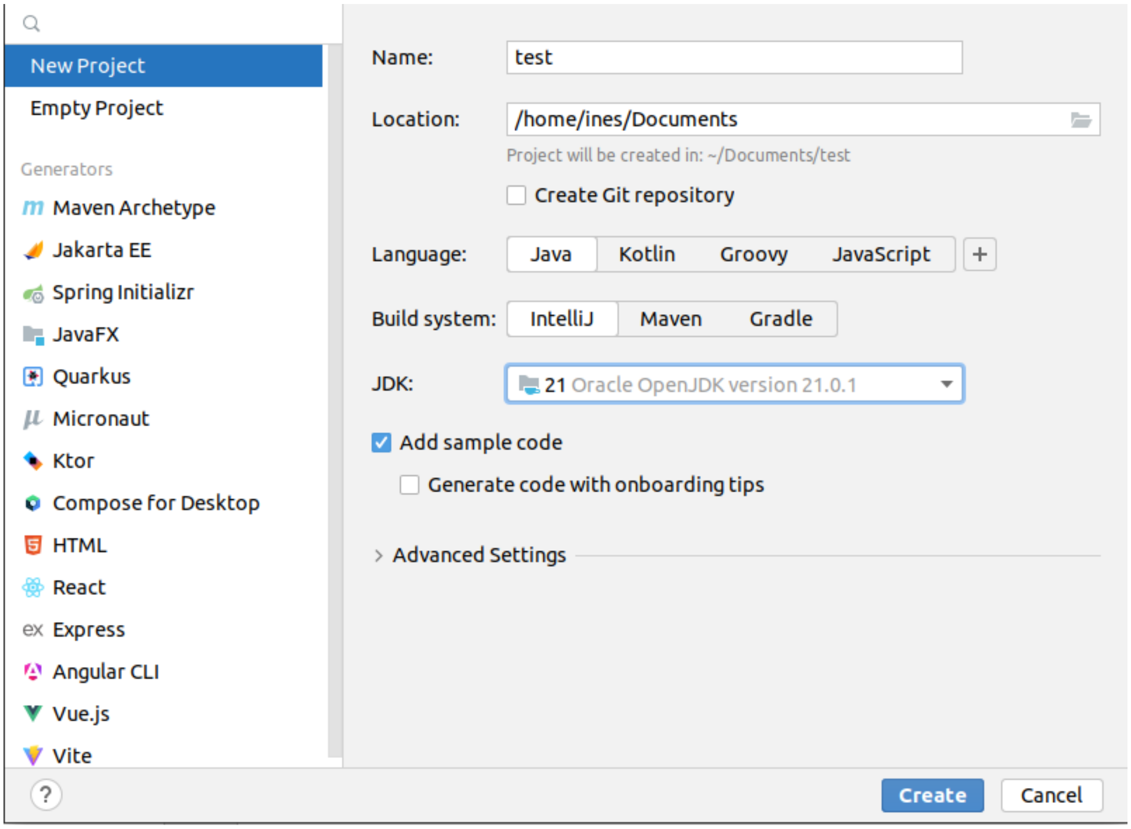
You might have already used an Integrated Development Environment (IDE) to write Python programs. In this course, we will do the same for programming in Java: we will use the free "Community Edition" of IntelliJ IDEA (simply referred to as "IntelliJ" hereafter). You can download the installer from https://www.jetbrains.com/idea/download/ (scroll down to find the free Community Edition, you don't need the commercial Ultimate Edition). Start the installer and follow the instructions.

The second thing you will need for Java programming is a *Java Development Kit* (JDK). A JDK is a software package that contains the tools that you need to build and run Java programs. The JDK also includes a *very, very large* library of useful classes for all kinds of programming tasks. You can see the content of the library here: https://docs.oracle.com/en/java/javase/21/docs/api/index.html.

Fortunately, IntelliJ can automatically download the JDK for you when you create a new project, so you don't have to worry about the JDK for now. But if later on you wish to write a Java application on a computer without IntelliJ, you'll have to manually download the JDK from https://openjdk.org/ and install it.

1.1.2 Creating a new project

Start IntelliJ. A window will open where you can create a new project. Click on the corresponding button and you should see a window like this one:



Name: test

Location: /home/ines/Documents

Project will be created in: ~/Documents/test

☐ Create Git repository

Language: Java | Kotlin | Groovy | JavaScript | +

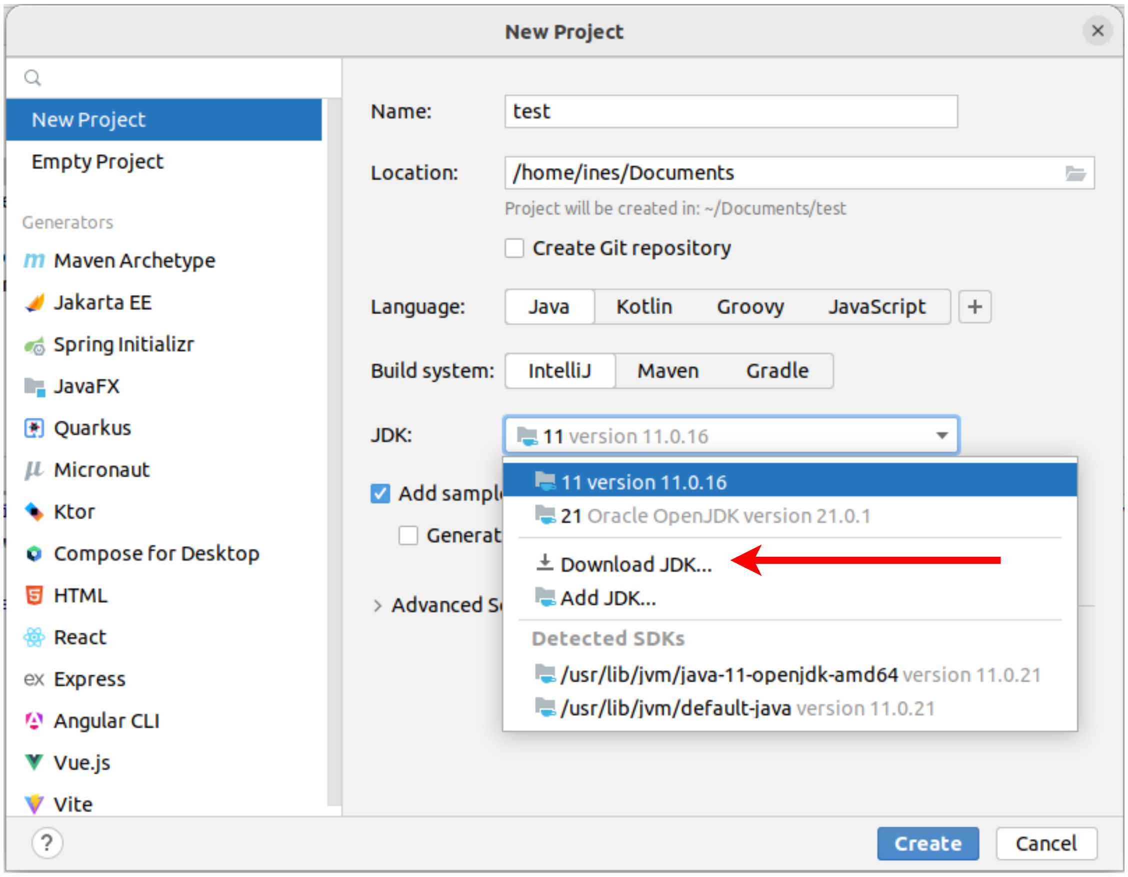
Build system: IntelliJ | Maven | Gradle

JDK: 21 Oracle OpenJDK version 21.0.1

☑ Add sample code

☐ Generate code with onboarding tips

To create a new project, you have to enter a project name (in the field `Name`) and a location on your disk where you want to store the project (in the field `Location`). Keep the other fields `Language`, `Build system`, and `Add sample code` as shown in the above picture. But there is something to do for the field `JDK`: as you can see in the picture, there was already JDK version 21 (and some other JDK versions) installed on my computer. If you have not already installed a JDK on your computer, open the dropdown list and choose `Download JDK...` as shown in the picture below:



Name: test

Location: /home/ines/Documents

Project will be created in: ~/Documents/test

□ Create Git repository

Language: Java

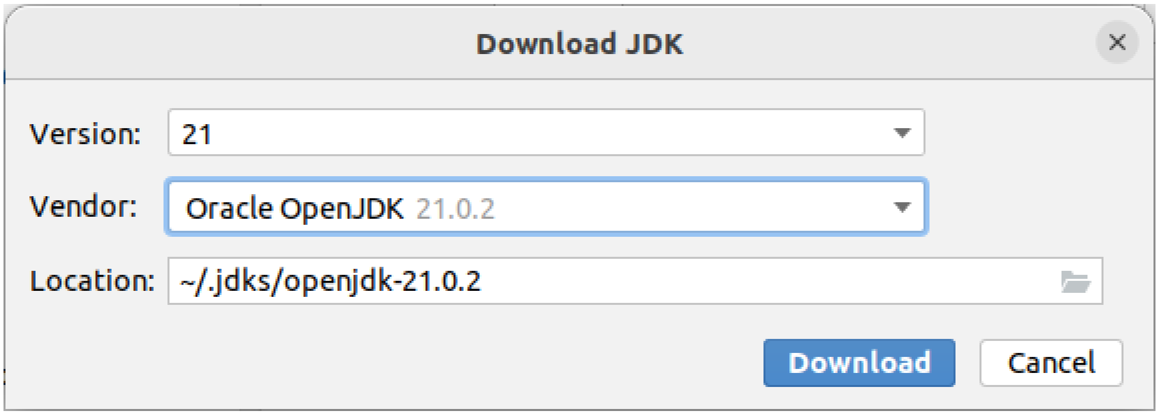
Build system: IntelliJ

JDK: 11 version 11.0.16

☑ Add sampl

□ Generat

A small window should appear where you can select which JDK version to download and install:

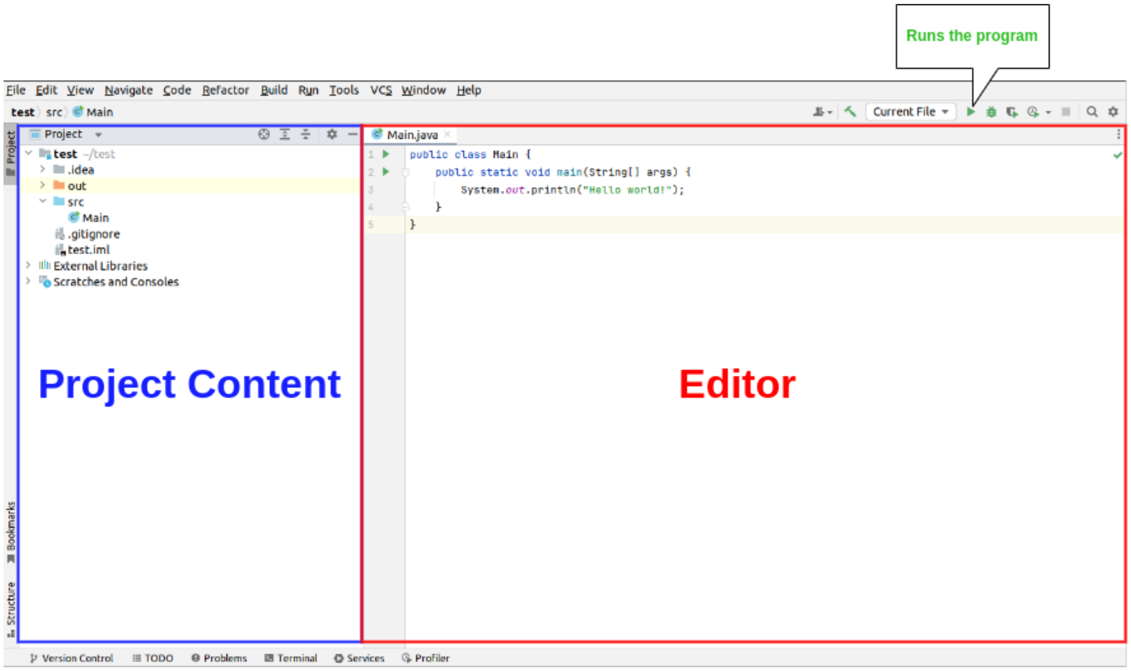


Version: 21

Vendor: Oracle OpenJDK 21.0.2

Location: ~/.jdks/openjdk-21.0.2

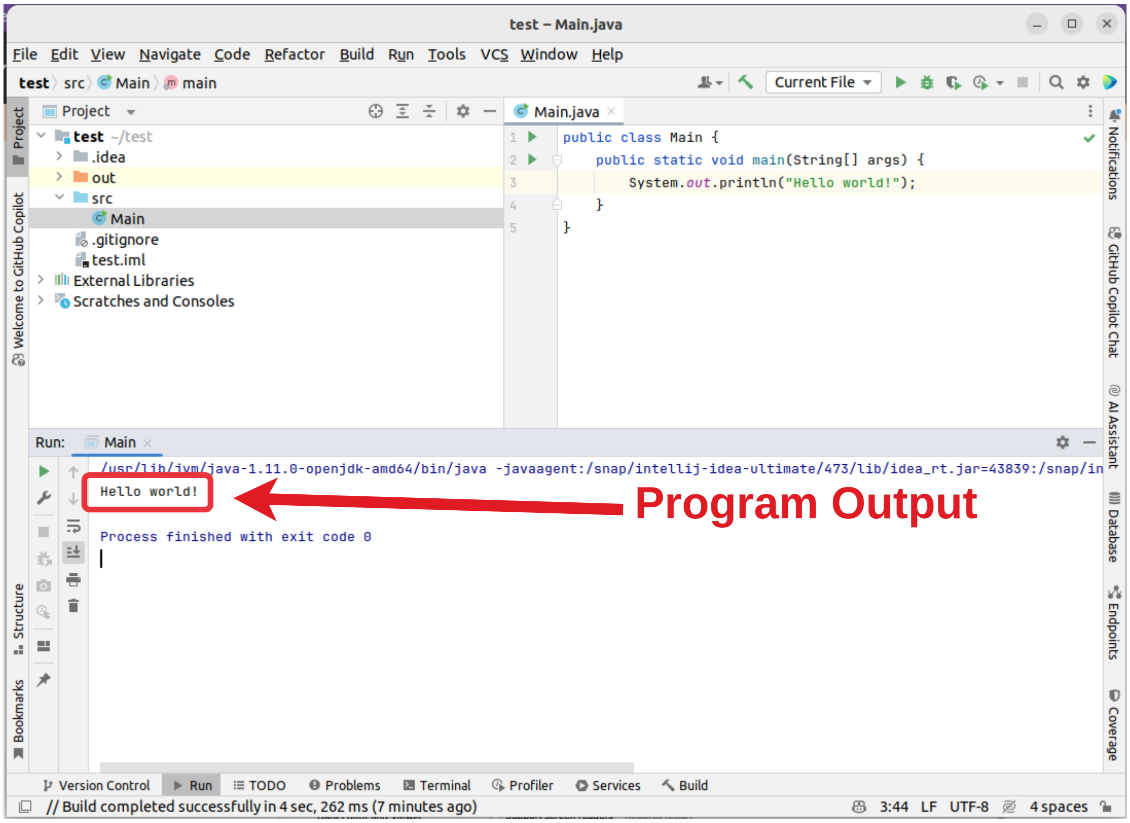
Select version 21 from the vendor `Oracle OpenJDK` (actually, any version newer than 17 is fine for this book). You can keep the location proposed by IntelliJ. Click the `Download` button and complete the JDK installation. Once everything is ready, you can finally create your first Java project. IntelliJ will normally automatically open the new project and show you the main window:



```java
public class Main {
    public static void main(String[] args) {
        System.out.println("Hello world!");
    }
}
```

In the left part of the window, you see the project structure. Since we have selected `Add sample code` in the project creation window, IntelliJ has already created a `src` directory with one file in it: `Main.java` (the file ending `.java` is not shown). When you double-click the file, its content is shown in the editor in the right part of the window.

Click on the right triangle in the upper right corner to start the program. A new view should appear at the bottom of the window with the output of the program:



```java
public class Main {
    public static void main(String[] args) {
        System.out.println("Hello world!");
    }
}
```

```
/usr/lib/jvm/java-1.11.0-openjdk-amd64/bin/java -javaagent:/snap/intellij-idea-ultimate/473/lib/idea_rt.jar=43839:/snap/in
Hello world!

Process finished with exit code 0
```

// Build completed successfully in 4 sec, 262 ms (7 minutes ago)

1.1.3 How do Java programs look like?

Here is the source code of the example program automatically created by IntelliJ in your project:

```java
public class Main {
    public static void main(String[] args) {
        System.out.println("Hello world!");
    }
}
```

And here is how an equivalent Python program would look like:

```python
print('Hello world!')
```

Why does the Java code look more complicated than the Python code? First of all, unlike Python, Java doesn't allow to write a statement like `print('Hello world!')` directly in a source code file. In Java, all statements MUST be inside a method and all methods MUST be inside a class. In our example, the statement `System.out.println("Hello world!")` is in the method `main()` and this method is in the class `Main`. Of course, a class in Java can have more than one method, and a Java program can contain more than one class.

You probably have already learned about classes and methods in Python and you might remember that classes are used to describe objects and methods are used to work with those objects. In our simple Java example, we don't need objects and all the complicated things that come with them (constructors, inheritance, etc.). The word `static` in the line `public static void main(String[] args)` indicates that the method `main()` behaves more like a traditional function in Python and not like a method for objects. In fact, no object is needed to execute a static method like `main()`. We will learn more about this later.

The second thing you might have noticed is the word `public` appearing twice in the first two lines of the code:

```java
public class Main {
    public static void main(String[] args) {
```

The word `public` in the first line indicates that the class `Main` can be used by others. It is not strictly necessary for this simple program and, in fact, our program will still work if you remove it (try it!). However, there is something important you have to know about public classes: If a class is marked as public, the source file that contains the class must have the same name as the class. That's the reason why the file is called `Main.java` and the public class in the file is called `Main` (Try to change the name of the class and see what happens!). Apart from that, the name `Main` for a class doesn't have any special meaning in Java. Our program would still work if we renamed the class to `Catweazle` or `Cinderella`, as long as we don't forget to rename the file as well. But note that **all class names in Java (public or not) start with an uppercase letter**.

The `public` in the second line is much more important for our example. A Java program can only be executed if it contains a method `main()` that is `public` *and* `static`. Remove the `public` or `static` from the second line and see what happens when you try to run the program. In general, **a Java program always starts at the public static main method**. If your program contains multiple classes with a main method, you have tell IntelliJ which one you want to start.

With this knowledge, can you guess what the following program prints?

```java
public class Main {
    static void printHello() {
        System.out.print("How do ");
        System.out.println("you do, ");
    }

    public static void main(String[] args) {
```

(continues on next page)

(continued from previous page)

```
        printHello();
        System.out.println("fellow kids?");
    }
}
```

(By the way, have you noticed the difference between `System.out.print` and `System.out.println`?)

A `.java` file can contain more than one class, however only one of these classes can be public. Here is the example from above with two classes:

```java
class MyOtherClass {
    static void printHello() {
        System.out.print("How do ");
        System.out.println("you do, ");
    }
}

public class Main {
    public static void main(String[] args) {
        MyOtherClass.printHello();
        System.out.println("fellow kids?");
    }
}
```

You can access the static content of a class from another class by using the name of the class, as demonstrated in the line `MyOtherClass.printHello()` in the example.

1.1.4 Types

You might already know that Python is a *strongly typed* language. That means that all “things” in Python have a specific type. You can see that by entering the following statements in the Python prompt:

```
>>> type("hello")
<class 'str'>
>>> type(1234)
<class 'int'>
>>> type(1234.5)
<class 'float'>
>>> type(True)
<class 'bool'>
```

Java is a strongly typed language, too. However, there is a big difference to Python: Java is also a *statically typed* language. We will not discuss all the details here, but in Java that means that most of the time you must indicate for *every* variable in your program what type of “things” it can contain.

Here is a simple Python program to calculate and print the area of a square:

```python
def calculateArea(side):
    return side * side

def printArea(message, side):
    area = calculateArea(side)
    print(message)
```

(continues on next page)

(continued from previous page)

```
    print(area)

t = 3 + 4
printArea("Area of square", t)
```

And here is the equivalent Java program:

```
public class Main {
    static int calculateArea(int side) {
        return side * side;
    }

    static void printArea(String message, int side) {
        int area = calculateArea(side);
        System.out.println(message);
        System.out.println(area);
    }

    public static void main(String[] args) {
        int t = 3 + 4;
        printArea("Area of square", t);
    }
}
```

Let's see what's going on with the types in the Java code:

- The line `int calculateArea(int side)` indicates that the method `calculateArea()` has a parameter `side` of type `int`. Furthermore, the `int` at the beginning of `int calculateArea(...` specifies that this method can only return a value of type `int`. This is called the *return type* of the method.
- The line `void printArea(String message, int side)` defines that the method `printArea()` has a parameter `message` of type `String` and a parameter `side` of type `int`. The method does not return anything, therefore it has the special return type `void`.
- Inside the method `printArea()`, we can see in the line `int area = calculateArea(side)` that the variable `area` has the type `int`.
- (Exercise for you: Look at the types that you can see in the `main()` method. We will explain later why that method always has a parameter `args`)

IntelliJ uses a special tool called the *Java compiler* that carefully verifies that there are no *type errors* in your program, i.e., that you have not made any mistakes in the types of the variables, method parameters, and return types in your program. Unlike Python, this *type checking* is done *before* your program is executed. You cannot even start a Java program that contains type errors!

Here are some examples that contain type errors. Can you find the mistakes?

- `int t = "Hello";`
- `boolean t = calculateArea(3);`
- `printArea(5, "Size of square");` (This example shows why it is easier to find bugs in Java than in Python)

1.2 The Java compiler and class files

In the previous section, we mentioned that a special tool, the *Java compiler*, checks your program for type errors. This check is part of another fundamental difference between Python and Java. Python is an *interpreted language*. That means that when you start a program written in Python in an IDE or on the command line with

```
> python myprogram.py
```

the Python-Interpreter will do the following things:

1. Load the file `myprogram.py`,
2. Do some checks to verify that your program doesn't contain syntax errors such as `print('Hello')))))`,
3. Execute your program.

Java, being a *compiled language*, works differently. To execute a Java program, there is another step done before your program can be executed:

1. First, the Java code has to be compiled. This is the job of the Java compiler, a tool that is part of the JDK. The compiler does two things:
   - It verifies that your source code is a well-formed Java program. This verification process includes the type checking described in the previous section.
   - It translates your Java source code into a more compact representation that is easier to process for your computer. This compact representation is called a *class file*. One such file will be created per class in your program. In IntelliJ, you can find the generated class files in the directory `out` in your project.
2. If the compilation of your code was successful, the *Java Virtual Machine* (JVM) is started. The JVM is a special program that can load and execute class files. The JVM doesn't need the source code (the `.java` files) of your program to execute it since the class files contain all the necessary information. When you are developing software for other people, it's usually the class files that you give to them, not the source code.

IntelliJ runs the Java compiler and starts the JVM for you when you press the green start button, but it's perfectly possible to do it by hand on the command line without an IDE:

```
> javac Main.java    # javac is the compiler and part of the JDK.
                     # It will generate the file Main.class

> java Main          # this command starts the JVM with your Main class
```

1.3 Basics

1.3.1 Primitive Types

As explained, Java requires that you specify the type of all variables (including method parameters) and the return types of all methods. Java differs between *primitive types* and complex types, such as arrays and objects. The primitive types are used for numbers (integers and real numbers), for Boolean values (`true` and `false`) and for single characters (`a`, `b`, etc.). However, there are several different number types. The below table shows all primitive types:

Type	Possible values	Example
`int`	$-2^{31}..2^{31}-1$	`int a = 3;`
`long`	$-2^{63}..2^{63}-1$	`long a = 3;`
`short`	$-2^{15}..2^{15}-1$	`short a = 3;`
`byte`	$-2^{7}..2^{7}-1$	`byte a = 3;`
`float`	$1.4*10^{-45}..3.4*10^{38}$	`float a = 3.45f;`
`double`	$4.9*10^{-324}..1.7*10^{308}$	`double a = 3.45;`
`char`	$0..2^{16}-1$	`char a = 'X';`
`boolean`	`true`, `false`	`boolean a = true;`

As you can see, each primitive type has a limited range of values it can represent. For example, a variable of type `int` can be only used for integer numbers between -2^{31} and $2^{31}-1$. If you don't respect the range of a type, very strange things will happen in your program! Try this code in IntelliJ (copy it into the `main()` method of your program):

```java
int a = 123456789;
int b = a * 100000;     // This is too large for the int type!
System.out.println(b);  // What will you get here?
```

For most examples in this book, it will be sufficient to use `int` (for integer numbers) and `float` (for real numbers). The types `long` and `double` provide a wider value range and more precision, but they are slower and your program will consume more memory when running.

Java supports the usual arithmetic operations with number types, that is `+` (addition), `-` (subtraction), `*` (multiplication), `/` (division), and `%` (modulo). There is also a group of operators that can be used to manipulate integer values on bit level (for example, left shift << and bitwise and &), but we will not discuss them further here.

The `char` type is used to work with individual characters (letters, digits,...):

```java
char c = 'a';
```

You might wonder why this type is shown in the above table as a type with values between 0 and 65535. This is because Java represents characters by numbers following a standard called *Unicode*. Consequently, you can do certain simple arithmetic operations with characters:

```java
char c = 'a';
c++;
System.out.println(c);  // prints 'b'
```

You can find more information about Unicode on https://en.wikipedia.org/wiki/Unicode.

1.3.2 Type casting

Java performs automatic conversions between values of different types if the destination type is "big" enough to hold the result. This is called *automatic type casting*. For this reason, these two statements are allowed:

```java
float a = 34;           // the int value 34 is casted to float 34.0f
float b = 6 * 4.5f;     // int multiplied by float gives float
```

But this is not allowed:

```java
int a = 4.5f;           // Error! float is not automatically casted to int
float b = 4.5f * 6.7;   // Error! float * double gives double
```

You can force the conversion by doing a *manual type cast*, but the result will be less precise or, in some situations, even wrong:

```
int a = (int) 4.5f;             // this will give 4
float b = (float) (4.5f * 6.7);
```

The Java class `Math` provides a large set of methods to work with numbers of different types. It also defines useful constants like `Math.PI`. Here is an example:

```
double area = 123.4;
double radius = Math.sqrt(area / Math.PI);

System.out.println("Area of disk: " + area);
System.out.println("Radius of disk: " + radius);
```

The complete documentation of the `Math` class can be found at https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/lang/Math.html.

1.3.3 What is a variable? A mental model

When working with variables of primitive types, you can imagine that every time your program reaches a line in your code where a variable is declared, the JVM will use a small part of the main memory (RAM) of your computer to store the value of the variable.

Java code	In memory during execution

`int a = 3;`
`int b = 4;`
	a: 3
	b: 4

When you assign the content of a variable to another variable, the value is copied:

Java code	In memory during execution
`a = b;`	a: 4 b: 4

The same also happens with the parameters of methods; when you call a method with arguments, for example `calculateArea(side)`, the argument values are copied into the parameter variables of the called method. Look at the following program and try to understand what it does:

```
public class Main {
    static void f(int x) {
        x = x + 1;
    }

    public static void main(String[] args) {
        int i = 3;
        f(i);
        System.out.println(i);
    }
}
```

The above program will print 3 because when you call the method f, the content of the variable i will be copied into the parameter variable x of the method. Even if the method changes the value of x with x = x + 1, the variable i will keep its value 3.

Note that it is illegal to use a local variable, i.e., a variable declared inside a method, before you have assigned a value to it:

```
public static void main(String[] args) {
    int a = 2;
    int b;
    int c;

    int d = a * 3;    // This is okay

    b = 3;
    int e = b * 3;    // This is okay

    int f = c * 3;    // Error! "c" has not been initialized.
}
```

1.3.4 Class variables

In our examples so far, all variables were either parameter variables or local variables of a method. Such variables are only "alive" when the program is inside the method during execution. However, you can also have variables that "live" outside a method. These variables are called *class variables* because they belong to a class, not to a method. Similar to static methods, we mark them with the keyword static:

```
public class Main {

    static int a = 3;   // this is a class variable

    static void increment() {
        a += 5;         // this is equivalent to  a = a + 5
    }

    public static void main(String[] args) {
        increment();
        System.out.println(a);
    }
}
```

In contrast to local variables, class variables do not need to be manually initialized. They are automatically initialized to 0 (for number types) or false (for the boolean type). Therefore, this code is accepted by the compiler:

```
public class Main {

    static int a;   //  is equivalent to  a = 0

    static void increment() {
        a += 5;
    }

    public static void main(String[] args) {
```

(continues on next page)

(continued from previous page)

```
        increment();
        System.out.println(a);
    }
}
```

Be careful when you have class variables and parameter or local variables with the same name:

```
public class Main {

    static int a = 3;

    static void increment(int a) {
        a += 5;     // this is the parameter variable
    }

    public static void main(String[] args) {
        increment(10);
        System.out.println(a);
    }
}
```

In the method `increment`, the statement `a += 5` will change the value of the parameter variable `a`, **not** of the class variable. We say that the parameter variable *shadows* the class variable because they have the same name. Inside the method `increment`, the parameter variable `a` has priority over the class variable `a`. We say that the method is the *scope* of the parameter variable.

In general, you should try to avoid shadowing because it is easy to make mistakes, but if you really need to do it for some reason, you should know that it is still possible to access the class variable from inside the scope of the parameter variable:

```
public class Main {

    static int a = 3;

    static void increment(int a) {
        Main.a += 5;   // we want the class variable!
    }

    public static void main(String[] args) {
        increment(10);
        System.out.println(a);
    }
}
```

1.3.5 Arrays (*fr.* tableaux)

If you need a certain number of variables of the same primitive type, it can be useful to use an array type instead. Arrays are similar to lists in Python. One big difference is that when you create a new array you have to specify its size, i.e., the number of elements in it:

```java
int[] a = new int[4];  // an array of integers with 4 elements
```

Another big difference is that all the elements of a Java array must have the same type, whereas a Python list can store elements of different types. In the example above, the Java array can only store `int` values.

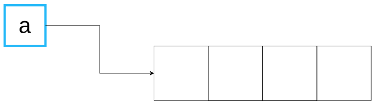
Once the array has been created, you can access its elements `a[0]`, `a[1]`, `a[2]`, `a[3]`. Like class variables, the elements of an array are automatically initialized when the array is created:

```java
int[] a = new int[4];   // all elements of the array are initialized to 0
a[2] = 5;
int b = a[1] + a[2];
System.out.println(b);  // prints "5" because a[1] is 0
```

Note that the size of an array is fixed. Once you have created it, you cannot change the number of elements in it. Unlike Python lists, arrays in Java do not have `slice()` or `append()` methods to add or remove elements. However, we will see later the more flexible `ArrayList` class.

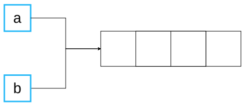
1.3.6 Mental model for arrays

There is an important difference between array variables and primitive-type variables. An array variable does not directly represent the array elements. Instead, an array variable can be seen as a *reference* to the content of the array. You can imagine it like this:

Java code	In memory during execution
`int[] a = new int[4];`	 **width** 50%

This difference becomes important when you assign an array variable to another array variable:

Java code	In memory during execution

`int[] a = new int[4];`
`int[] b = a;`
	

In that case, **only the reference to the array is copied, not the array itself**. This means that both variables `a` and `b` are now referencing the same array. This can be shown with the following example:

```java
int[] a = new int[4];
int[] b = a;              // a and b are now references to the same array
b[2] = 5;
System.out.println(a[2]); // prints "5"
```

This also works when you give an array as an argument to a method:

```java
public class Main {

    static void five(int[] x) {
        x[2] = 5;
    }

    public static void main(String[] args) {
        int[] a = new int[4];
        five(a);
        System.out.println(a[2]);  // prints "5"
    }
}
```

In this example, the method `five()` receives a *reference* to the array `a` (i.e., not a copy of it), which allows the method to modify the content of the array `a`.

1.3.7 Initializing an array

There is a convenient way to create and initialize an array in one single step:

```java
int[] a = { 2, 5, 6, -3 };  // an array with four elements
```

This is equivalent to the longer code:

```java
int[] a = new int[4];  // Creation of the array

// Initialization of the array
a[0] = 2;
a[1] = 5;
a[2] = 6;
a[3] = -3;
```

But note that this short form is only allowed when you initialize a newly declared array variable. If you want to create a new array and assign it to an existing array variable, you have to use a different syntax:

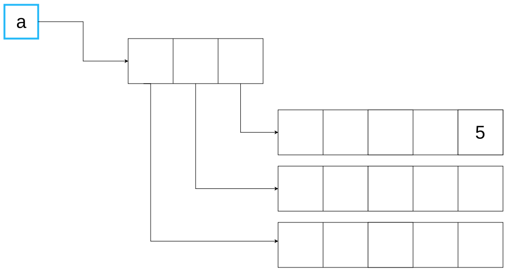
```java
int[] a = { 2, 5, 6, -3 }
a = new int[]{ 1, 9, 3, 4 };
```

1.3.8 Multi-dimensional arrays

Arrays can have more than one dimension. For example, two-dimensional arrays are often used to represent matrices in mathematical calculations:

```
int[][] a = new int[3][5];  // this array can be used to represent a 3x5 matrix
a[2][4] = 5;
```

You can imagine a two-dimensional array as an array where each element is again a reference to an array:



An `int[3][5]` is therefore an array of three arrays containing five elements each. The following code illustrates this:

```
int[][] a = new int[3][5];
int b[] = a[0];  // b is now a reference to an int array with 5 elements
b[3] = 7;
System.out.println(a[0][3]);  // b[3] and a[0][3] are the same element
```

Again, there is a convenient way to create and initialize multi-dimensional arrays in one step:

```
// 3x3 Identity matrix from the Linear Algebra course
int[][] a = {
    { 1, 0, 0 },
    { 0, 1, 0 },
    { 0, 0, 1 }
};
```

1.3.9 Partially initialized arrays

It is possible to create a “partially initialized” two-dimensional array in Java:

```
int[][] a = new int[3][];
```

Again, this is an array of arrays. However, because we have only specified the size of the first dimension, the elements of this array are initialized to `null`. We can initialize them later:

```
int[][] a = new int[3][];
a[0] = new int[5];          // 5 elements
a[1] = new int[5];          // 5 elements
a[2] = new int[2];          // 2 elements. That's allowed!
```

As shown in the above example, the elements of a multi-dimensional array are all arrays, but they do not need to have the same size.

1.3.10 Arrays and class variables

Array variables can be class variables (with the `static` keyword), too. If you don't provide an initial value, the array variable will be initialized with the value `null`:

```
public class Main {

  static int[] a;   //  automatically initialized to null

  public static void main(String[] args) {
      // this compiles, but it gives an error during execution,
      // because we have not initialized a
      System.out.println(a[2]);
  }
}
```

You can think of the value `null` as representing an invalid reference.

1.3.11 "While" loops

The two most common loop constructs in Java are the `while` loop and the `for` loop.

The `while` loop in Java is very similar to its namesake in Python. It repeats one or more statements (we call them the *body* of the loop) as long a condition is met. Here is an example calculating the sum of the numbers from 0 to 9 (again, the surrounding `main()` method is not shown):

```
int sum = 0;
int i = 0;
while (i<10) {
    sum += i;    // this is equivalent to sum = sum + i
    System.out.println("Nearly there");
    i++;         // this is equivalent to i = i + 1
}
System.out.println("The sum is " + sum);
```

Warning: The two statements inside the `while` loop must be put in curly braces `{...}`. If you forget the braces, only the *first* statement will be executed by the loop, independently of how the line is indented:

```
int sum = 0;
int i = 0;
while (i<10)                             // oops, we forgot to put a brace '{' here!
    sum += i;                            // this statement is INSIDE the loop
    System.out.println("Nearly there");  // this statement is OUTSIDE the loop!!!
    i++;                                 // this statement is OUTSIDE the loop!!!

System.out.println("The sum is " + sum);
```

This is also true for other types of loops and for if/else statements.

To avoid "accidents" like the one shown above, it is highly recommended to always use braces for the body of a loop or if/else statement, even if the body only contains one statement.

1.3.12 Simple “for” loops

There are two different ways for loops can be used. The simple for loop is often used to do something with each element of an array or list (We will learn more about lists later):

```
int[] myArray = new int[]{ 2, 5, 6, -3 };
int sum = 0;
for (int elem : myArray) {
    sum += elem;
}
System.out.println("The sum is " + sum);
```

The for loop will do as many iterations as number of elements in the array, with the variable elem successively taking the values of the elements.

1.3.13 Complex “for” loops

There is also a more complex version of the for loop. Here is again our example calculating the sum of the numbers from 0 to 9, this time with a for loop:

```
int sum = 0;
for (int i = 0; i<10; i++) {
    sum += i;
    System.out.println("Nearly there");
}
System.out.println("The sum is " + sum);
```

The first line of the for loop consists of three components:

1. a statement that is executed when the loop starts. In our example: `int i = 0`.
2. an expression evaluated *before* each iteration of the loop. If the expression is `false`, the loop stops. Here: `i<10`.
3. a statement that is executed *after* each iteration of the loop. Here: `i++`.

The complex for loop is more flexible than the simple version because it gives you full control over what is happening in each iteration. Here is an example where we calculate the sum of every second element of an array:

```
int[] myArray = new int[]{ 2, 5, 6, -3, 4, 1 };
int sum = 0;
for (int i = 0; i<myArray.length; i += 2) {
    sum += myArray[i];
}
System.out.println("The sum is " + sum);
```

In this example, we have done two new things. We have used myArray.length to get the size of the array myArray. And we have used the statement i+=2 to increase i by 2 after each iteration.

1.3.14 Stopping a loop and skipping iterations

Like in Python, you can leave any loop with the `break` statement:

```
int sum = 0;
for (int i = 0; i<10; i++) {
    sum += i;
    if (sum>5) {
        break;
    }
}
```

And we can immediately go to the next iteration with the `continue` statement:

```
int sum = 0;
for (int i = 0; i<10; i++) {
    if (i==5) {
        continue;
    }
    sum += i;
}
```

But you should only use `break` and `continue` if they make your program easier to read. In fact, our above example was not a good example because you could just write:

```
for (int i = 0; i<10; i++) {
    if (i!=5) {     // easier to understand than using "continue"
        sum += i;
    }
}
```

1.3.15 "If/else" statements

As you have seen in some of the examples above, Java has an `if` statement that is very similar to the one in Python. Here is an example that counts the number of negative and positive values in an array:

```
int[] myArray = new int[]{ 2, -5, 6, 0, -4, 1 };
int countNegative = 0;
int countPositive = 0;
for(int elem : myArray) {
    if(elem<0) {
        countNegative++;
    }
    else if(elem>0) {
        countPositive++;
    }
    else {
        System.out.println("Value zero found");
    }
}
System.out.println("The number of negative values is " + countNegative);
System.out.println("The number of positive values is " + countPositive);
```

As with loops, be careful not to forget to use curly braces `{...}` if the body of the if/else statement contains more than one statement. **It is highly recommended to always use braces, even if the body contains only one statement.**

1.3.16 Comparison and logical operators

The `if` statement requires a Boolean expression, i.e., an expression that evaluates to `true` or `false`. There are several operators for Boolean values that are quite similar to the ones you know from Python:

```
boolean b1 = 3 < 4;     // we also have <, >, <=, >=, ==, !=
boolean b2 = !b1;       // "not" in Python
boolean b3 = b1 && b2;  // "and" in Python
boolean b4 = b1 || b2;  // "or" in Python
```

1.3.17 “Switch” statement

Imagine a program where you test a variable for different values:

```
// two integer variables that represent our position on a map
int x = 0, y = 0;

// the directions in which we want to go
char[] directions = new char[]{'N', 'S', 'S', 'E', 'E', 'W'};

// let's go!
for (char c : directions) {
    if (c == 'N') {
        y++;            // we go North
    }
    else if (c == 'S') {
        y--;            // we go South
    }
    else if (c == 'W') {
        x--;            // we go West
    }
    else if (c == 'E') {
        x++;            // we go East
    }
    else {
        System.out.println("Unknown direction");
    }
    System.out.println("The new position is " + x + " , " + y);
}
```

Java has a `switch` statement that allows you to write the above program in a clearer, more compact way:

```
int x = 0, y = 0;

char[] directions = new char[]{'N', 'S', 'S', 'E', 'E', 'W'};

for (char c : directions) {
    switch (c) {
        case 'N' -> { y++; }     // we go North
```

(continues on next page)

(continued from previous page)

```
        case 'S' -> { y--; }     // we go South
        case 'W' -> { x--; }     // we go West
        case 'E' -> { x++; }     // we go East
        default -> { System.out.println("Error! Unknown direction"); }
    }
    System.out.println("The new position is " + x + " , " + y);
}
```

Note that the above code only works with Java version 14 or newer. In older Java versions, the `switch` statement is a bit more complex as it necessitates to separate the cases using the `break` statement:

```
switch (c) {
    case 'N':
        y++;
        break;  // if you forget the "break", very bad things will happen!
    case 'S':
        y--;
        break;
    case 'W':
        x--;
        break;
    case 'E':
        x++;
        break;
    default:
        System.out.println("Error! Unknown direction");
}
```

Since Java 8 is still widely used, you should familiarize yourself with both versions of the `switch` statement.

1.3.18 Strings

Variables holding string values have the type `String`. Strings can be concatenated to other strings with the + operator. This also works for primitive types:

```
String s1 = "This is a string";
String s2 = "This is another string";
String s3 = s1 + "---" + s2 + 12345;
System.out.println(s3);
```

The `String` class defines many interesting methods that you can use to work with strings. If you check the documentation at https://docs.oracle.com/en/java/javase/21/docs//api/java.base/java/lang/String.html, you will notice that some methods of the `String` class are static and some are not. For example, the static method `valueOf` transforms a number value into a string:

```
double x = 1.234;
String s = String.valueOf(x);
System.out.println(s);
```

But most methods of the `String` class are not static, i.e., you have to call them on a string value or string variable. Here are some frequently used methods:

```
String s = "Hello world";
int l = s.length();                 // the length of the string
boolean b = s.isEmpty();            // true if the string has length 0
char c = s.charAt(3);               // the character in the string at position 3
boolean b2 = s.startsWith("Hello"); // true if the string starts with "Hello"
int i = s.indexOf("wo");            // gives the position of "wo" in the string
String t = s.substring(2);          // the string starting at position 2
```

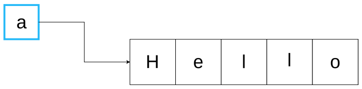
There are also some methods for strings that are located in other classes. The most useful ones are the methods to convert strings to numbers. For `int` values, there is for example the static method `parseInt` in the `Integer` class:

```
int i = Integer.parseInt("1234");
```

Similar methods exist in the classes `Long`, `Float`, `Double`, etc. for the other primitive types. All these classes are defined in the package `java.lang`, for which you can find the documentation at https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/lang/package-summary.html.

1.3.19 Mental model for strings

Like array variables, string variables are references to the content of the string:

Java code	In memory during execution
`String a = "Hello";`	

1.3.20 Comparing things

Primitive-type values can be tested for equality with the == operator:

```
int i = 3;
if(i == 3) {
    System.out.println("They are the same!");
}
```

However, **this will not work for arrays or strings**. Indeed, since array and string variables only contain references, the == operator will compare the *references*, not the *content* of the arrays or strings! The following example shows the difference:

```
int i = 3;
System.out.println(i == 3);     // true. Primitive type.

int[] a = {1, 2, 3};
int[] b = {1, 2, 3};
System.out.println(a == b);     // false. Two different arrays.

int[] c = a;
System.out.println(a == c);     // true. Same reference.

String s1 = "Hello" + String.valueOf(1234);
```

(continues on next page)

(continued from previous page)

```
String s2 = "Hello1234";
System.out.println(s1 == s2);   // false. Two different strings.
```

Comparing arrays or strings with == is a very common mistake in Java. Be careful!

To compare the *content* of two strings, you must use their `equals()` method:

```
String s1 = "Hello" + String.valueOf(1234);
String s2 = "Hello1234";
System.out.println(s1.equals(s2));   // true
```

There is also an `equals()` method to compare the content of two arrays, but it is a static method of the class `Arrays` in the package `java.util`. To use this class, you have to import it into your program. Here is the complete code:

```
import java.util.Arrays;

public class Main {
    public static void main(String[] args) {
        int[] a = {1, 2, 3};
        int[] b = {1, 2, 3};
        System.out.println(Arrays.equals(a, b));  // true
    }
}
```

The `Arrays` class contains many useful methods to work with arrays, such as methods to set all elements of an array to a certain value, to make copies of arrays, or to transform an array into a string. See the documentation at https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/util/Arrays.html.

You might wonder why we need the line `import java.util.Arrays` but we didn't need to import the classes `Math`, `Integer` or `String` in our other examples. That's because those classes are in the package `java.lang`, which is the only package that is automatically imported by the Java compiler.

1.3.21 Enumerations

Enums in Java are a type that represents a group of constants (unchangeable variables, like final variables). They are a powerful mechanism for defining a set of named values, which you can use in a type-safe way. Enums are a feature that enhances code readability and maintainability by allowing you to declare collections of constants with their own namespace.

Using a `switch` statement is very convenient for reacting according to the value of an enum variable. Alternatively, since they are constants and there's only one instance of each enum constant in the JVM, you can use the == operator to compare them for equality.

We revisit the direction instruction example but using an `enum` instead of `char` to encode the directions. Note that this code is safer since it is impossible to have a direction that is not in the list of the ones defined in the enum. Therefore we don't have to deal with the possibility of an unknown direction in the `switch` statement.

```
public class DirectionFollower {

    public enum Direction {
        NORTH, EAST, SOUTH, WEST;
    }

    /**
```

(continues on next page)

(continued from previous page)

```
     * Computes the final coordinates after applying a series of movements to a starting␣
↪position.
     *
     * @param start The starting coordinates as an array of size two, where start[0] is␣
↪the x-coordinate and start[1] is the y-coordinate.
     * @param directions An array of {@code Direction} enums that represent the sequence␣
↪of movements to apply to the starting coordinates.
     * @return A new array of size two representing the final coordinates.
     *
     * Example:
     * {@code
     * int[] start = {0, 0};
     * Direction[] directions = {Direction.NORTH, Direction.EAST, Direction.NORTH,␣
↪Direction.WEST};
     * int[] finalCoordinates = followDirections(start, directions);
     * // This will yield final coordinates of [0, 2]
     * }
     */
    public static int[] followDirections(int[] start, Direction[] directions) {
        int[] result = new int[]{start[0], start[1]};

        for (Direction direction : directions) {
            switch (direction) {
                case NORTH:
                    result[1]++;
                    break;
                case EAST:
                    result[0]++;
                    break;
                case SOUTH:
                    result[1]--;
                    break;
                case WEST:
                    result[0]--;
                    break;
            }
        }
        return result;
    }

    public static void main(String[] args) {
        int[] start = {0, 0};
        Direction[] directions = {
                Direction.NORTH,
                Direction.EAST,
                Direction.EAST,
                Direction.SOUTH,
                Direction.WEST,
                Direction.NORTH,
                Direction.NORTH
        };
```

(continues on next page)

(continued from previous page)

```
        int[] finalCoordinates = followDirections(start, directions);
        System.out.println("The final coordinates are: [" + finalCoordinates[0] + ", " +␣
↪finalCoordinates[1] + "]");
    }
}
```

1.4 Exceptions

In Java, there are two ways to exit a method: by using the `return` statement or by *throwing an exception*. You already know the `return` statement, so in the following we explain how exceptions work.

1.4.1 Throwing an exception

Exceptions are a mechanism for stopping the execution of a method when an exceptional situation occurs that deviates from how the method is normally used. To do this, the `throw` statement is used. Typically, you give the statement an instance of the class `Exception` (or one of its subclasses) that contains information about why the exception was thrown:

```
class Employee {
    Employee boss;

    void setBoss(Employee boss) throws Exception {
        if(this == boss) {
            throw new Exception("An employee cannot be their own boss");
        }
        else {
            this.boss = boss;
        }
    }
}
```

In general, a method that can throw an exception must indicate this in the method declaration with the keyword `throws` and the class of the thrown exception object.

When a method calls a method that can throw an exception, it can react to an exception by catching it. To do this, it must put a `try-catch` block around the calls of the method.

```
public class Main {
    public static void main(String[] args) {
        Employee peter = new Employee();
        Employee anna = new Employee();

        try {
            peter.setBoss(anna);    // this is okay
            peter.setBoss(peter);   // this will throw an exception
        }
        catch(Exception e) {
            System.out.println("An exception happened: " + e.getMessage());
        }
```

(continues on next page)

(continued from previous page)

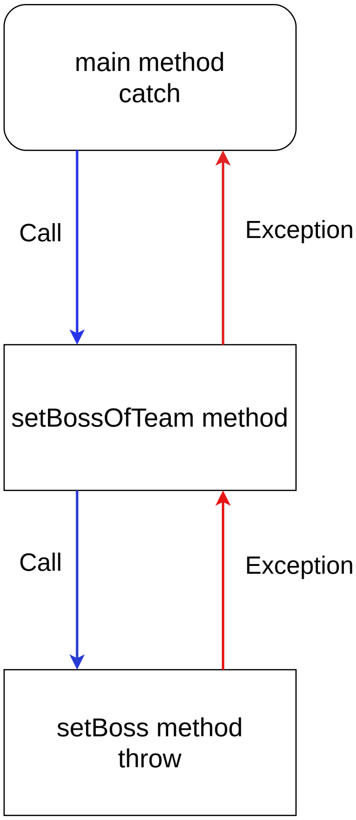
```java
    }
}
```

When the `setBoss()` method throws an exception, the execution of the code will directly go to the statement(s) specified inside the `catch` block. We say that the message is “caught”. The variable `e` contains a reference to the `Exception` object specified in the `throw` statement.

What makes exceptions interesting is that the caller method can decide to not catch the exception. In that case, the exception will be passed to the method that called the caller method and so on until the exception is caught. This is illustrated in the following example:

```java
public class Main {
    static void setBossOfTeam(Employee[] team, Employee boss) throws Exception {
        for(Employee employee : team) {
            employee.setBoss(boss);    // setBoss(...) can throw an exception,
                                       // but we don't catch it here
        }
    }

    public static void main(String[] args) {
        Employee peter = new Employee();
        Employee anna = new Employee();

        try {
            // a team with two employees:
            Employee team[] = { peter, anna };
            setBossOfTeam(team, peter);  // this will throw an exception
        }
        catch(Exception e) {
            System.out.println("An exception happened: " + e.getMessage());
        }
    }
}
```

In the above example, the `main()` method calls the `setBossOfTeam()` method which then calls the `setBoss()` method. The `setBossOfTeam()` method does not catch any exceptions. This means that if an exception is thrown in `setBoss()`, the exception will be passed to `main()` where it is caught, as shown below:



1.4.2 Using Exception subclasses

By creating subclasses of the `Exception` class, we can help the method that catches the exception to understand why the exception happened:

```
class SelfBossException extends Exception {
    SelfBossException(String message) {
        super(message);
    }
}

class NoBossException extends Exception {
    NoBossException(String message) {
        super(message);
    }
}

class Employee {
    Employee boss;

    void setBoss(Employee boss) throws SelfBossException, NoBossException {
        if (this == boss) {
            throw new SelfBossException("An employee cannot be their own boss");
        } else if (boss == null) {
```

(continues on next page)

(continued from previous page)

```java
            throw new NoBossException("You cannot take the boss away from an employee");
        } else {
            this.boss = boss;
        }
    }
}

public class Main {
    public static void main(String[] args) {
        Employee peter = new Employee();
        Employee anna = new Employee();

        try {
            peter.setBoss(anna);
            peter.setBoss(null);  // this will throw a NoBossException
        } catch (SelfBossException e) {
            System.out.println("SelfBossException happened: " + e.getMessage());
        } catch (NoBossException e) {
            System.out.println("NoBossException happened: " + e.getMessage());
        }
    }
}
```

If we don't want to use separate `catch` blocks for the different `Exception` subclasses, we can write the catch statement also like this:

```java
public static void main(String[] args) {
    Employee peter = new Employee();
    Employee anna = new Employee();

    try {
        peter.setBoss(anna);
        peter.setBoss(null);  // this will throw a NoBossException
    }
    catch(SelfBossException | NoBossException e) {
        System.out.println("Some exception happened: " + e.getMessage());
    }
}
```

And if we want to catch all exceptions (not only `SelfBossException` and `NoBossException`), we can still write:

```java
public static void main(String[] args) {
    Employee peter = new Employee();
    Employee anna = new Employee();

    try {
        peter.setBoss(anna);
        peter.setBoss(null);  // this will throw a NoBossException
    }
    catch(Exception e) {
        System.out.println("Some exception happened: " + e.getMessage());
    }
}
```

The above code works, because a statement like `catch(XYZ e) { ... }` catches all exceptions of the class `XYZ` **and of any subclass** of `XYZ` if the try-catch block has no other `catch` statement for a specific subclass of `XYZ`.

1.4.3 Checked vs unchecked exceptions

The exceptions that we threw in the above examples are all *checked exceptions*. This means that the compiler verifies that the exceptions are correctly declared in the `throws` part of the method declaration if the method does not catch them.

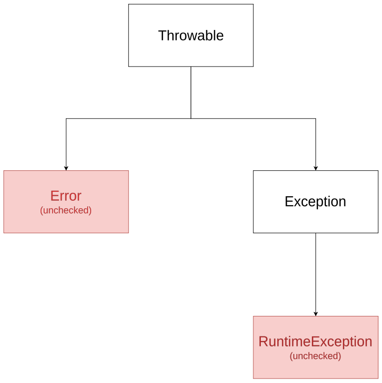
However, there are some exceptions for which the compiler does not perform this verification. Such exceptions are called *unchecked*. A famous unchecked exception is the `NullPointerException` that is thrown by the JVM when a program tries to access a null reference:

```
public class Main {
    public static void main(String[] args) {
        Object obj = null;
        String s = obj.toString();   // this will throw a NullPointerException
    }
}
```

As you can see in the above example, no `throws` declaration or `try-catch` block is required for an unchecked exception, but you can still catch it if you want:

```
public class Main {
    public static void main(String[] args) {
        Object obj = null;

        try {
            String s = obj.toString();
        }
        catch(NullPointerException e) {
            System.out.println("A null pointer exception happened!");
        }
    }
}
```

Unchecked exceptions are either instances of the class `Error` or of the class `RuntimeException` (or of a subclass of these classes). `RuntimeException` is a subclass of `Exception`, and `Error` and `Exception` are subclasses of the class `Throwable`. All instances of `Throwable` (or of a subclass of that class) can be thrown with a `throw` statement. The class hierarchy for these classes is shown below:



1.4.4 Do we need exceptions?

Strictly speaking, you *do not need* exceptions. For our example, our `setBoss` method from above

```java
class Employee {
    Employee boss;

    // throws an exception if there is an error
    void setBoss(Employee boss) throws Exception {
        if(this == boss) {
            throw new Exception("An employee cannot be their own boss");
        }
        else {
            this.boss = boss;
        }
    }
}
```

could be written without exceptions:

```java
class Employee {
    Employee boss;

    // returns false if there is an error
    boolean setBoss(Employee boss) {
        if(this == boss) {
            return false;
        }
        else {
            this.boss = boss;
            return true;
        }
    }
}
```

Consequently, we would not need to catch the exception when we call the method:

```java
public class Main {
    public static void main(String[] args) {
        Employee peter = new Employee();
        Employee anna = new Employee();

        boolean success = peter.setBoss(anna);
        if(success) {
            success = peter.setBoss(peter);
        }
        if(!success) {
            System.out.println("Something bad happened");
        }
    }
}
```

As you can see above, the code becomes more complicated without exceptions since we have to check the result of every call of `setBoss()`. However, we should also mention here that programs without exceptions are easier to understand. Look at these two lines of code in the version of the main method with exceptions:

```
peter.setBoss(anna);
peter.setBoss(null);
```

Just by reading these two lines, it is not obvious that the second call to `setBoss` is not executed if the first call detects a problem. In the version without exceptions this is immediately clear:

```
boolean success = peter.setBoss(anna);
if(success) {
    success = peter.setBoss(peter);
}
```

For this reason, exceptions should only be used sparingly. Fortunately, in many program, you don't need to throw your own exceptions, and often the only place you need to catch an exception is when using the existing I/O classes of the JDK. We will show an example in the next section.

1.4.5 Exceptions and I/O operations

The JDK provides many classes that help you to work with files and communicate with other computers in the Internet. For example, the package `java.io` contains classes to read data from files, to create new files, to delete files, etc. Many of the methods of these classes throw an instance of the `IOException` class if they encounter a problem.

The below example reads two characters from a text file:

```
import java.io.FileReader;
import java.io.IOException;

public class Main {
    public static void main(String[] args) {
        try {
            // open the file "somefile.txt"
            FileReader reader = new FileReader("somefile.txt");

            // read two characters from the file
            char c1 = (char) reader.read();
            char c2 = (char) reader.read();

            // close the file
            reader.close();
        }
        catch(IOException e) {
            System.out.println(e);
        }
    }
}
```

The constructor of the `FileReader` class throws a `FileNotFoundException` if the specified file `somefile.txt` does not exist. The `read()` method throws an `IOException` if there was a problem when reading the file, for example because the user was not allowed to read that file. Since `FileNotFoundException` is a subclass of `IOException`, we can catch both exceptions with a single `catch(IOException e) {...}`.

The above code has a weakness: If the `read()` method throws an exception, the line `reader.close()` is not executed and the file is not closed. The following code solves this problem by using a `finally` block:

```
import java.io.FileReader;
import java.io.IOException;

public class Main {
    public static void main(String[] args) {
        try {
            // open the file "somefile.txt"
            FileReader reader = new FileReader("somefile.txt");

            try {
                // read two characters from the file
                char c1 = (char) reader.read();
                char c2 = (char) reader.read();
            }
            finally {
                // close the file
                reader.close();
            }
        }
        catch(IOException e) {
            System.out.println(e);
        }
    }
}
```

The JVM *always* executes the statements in a `finally` block after the preceding `try` block, even if an exception happened inside the `try` block or the `try` block contains a return statement. For this reason `finally` blocks are often used in combination with try/catch blocks to “clean up” used resources (e.g., close a file).

The above situation (opening a file, using it, and then closing it) is very common in Java programs. For this reason, Java has a special compact form of the `try` block that is equivalent to the above program. When we us this special form, the Java compiler automatically adds the `finally` block and the `reader.close()` statement to our program:

```
import java.io.FileReader;
import java.io.IOException;

public class Main {
    public static void main(String[] args) {
        try(FileReader reader = new FileReader("somefile.txt")) {
            char c1 = (char) reader.read();
            char c2 = (char) reader.read();
        }
        catch(IOException e) {
            System.out.println(e);
        }
    }
}
```

CHAPTER

TWO

OBJECT-ORIENTED PROGRAMMING

2.1 Basics

2.1.1 Creating your own objects

Computer programs are about organizing data and working with that data. In some applications, the primitive types, arrays, and strings are enough, but often you have data that is more complex than that. For example, imagine a program to manage employees in a company. We can describe the fact that each employee has a name and a salary, by defining a new *class* in our Java program:

```java
class Employee {
    String name;    // the name of the employee
    int salary;     // the salary of the employee
}
```

Classes allow us to create new *objects* from them. In our example, each object of the class Employee represents an employee, which makes it easy to organize our data:

```java
class Employee {
    String name;
    int salary;
}

public class Main {
    public static void main(String[] args) {
        Employee person1 = new Employee();    // a new object!
        person1.name = "Peter";
        person1.salary = 42000;

        Employee person2 = new Employee();    // a new object!
        person2.name = "Anna";
        person2.salary = 45000;

        int salaryDifference = person1.salary - person2.salary;
        System.out.println("The salary difference is " + salaryDifference);
    }
}
```

The two objects that we created and put into the local variables person1 and person2 are called *instances* of the class Employee, and the two variables name and age are called *instance variables* of the class Employee. Since they are not static, they belong to the instances, and each instance has its own name and age.

2.1.2 Initializing objects

In the above example, we first created the object, and then set the values of its instance variables:

```
Employee person1 = new Employee();
person1.name = "Peter";
person1.salary = 42000;
```

Like static variables, instance variables are automatically initialized with the value 0 (for number variables), with `false` (for Boolean variables), or with `null` (for all other types). In our example, this is dangerous because we could forget to specify the salary of the employee:

```
Employee person1 = new Employee();
person1.name = "Peter";
// oops, the salary is 0
```

There are several ways to avoid this kind of mistake. One way is to initialize the variable in the class definition:

```
class Employee {
    String name;
    int salary = 10000;
}
```

Of course, this is only useful if you want that all employees start with a salary of 10000. The other way is to define a *constructor* in your class. The constructor is a special method that has the same name as the class. It can have parameters but it has no return type:

```
class Employee {
    String name;
    int salary;

    // the constructor
    Employee(String n, int s) {
        this.name = n;
        this.salary = s;
    }
}
```

If you provide a constructor for your class, the Java compiler will verify that you use it to create new objects:

```
Employee person1 = new Employee("Peter", 42000);
// Okay. We have now a new employee with
//    person1.name "Peter"
//    person1.salary 42000

Employee person2 = new Employee();   // not allowed. You must use the constructor!
```

In our example, the constructor took two parameters `n` and `s` and used them to initialize the instance variables `name` and `salary` of a new `Employee` object. But how does the constructor know which object to initialize? Do we have to tell the constructor that the new object is in the variable `person1`? Fortunately, it's easier than that. The constructor can always access the object being constructed by using the keyword `this`. Therefore, the line

```
this.name = n;
```

means that the instance variable `name` of the new object will be initialized to the value of the parameter variable `n`. We could even use the same names for the parameter variables and for the instance variables:

```
class Employee {
    String name;
    int salary;

    Employee(String name, int salary) {
        this.name = name;
        this.salary = salary;
    }
}
```

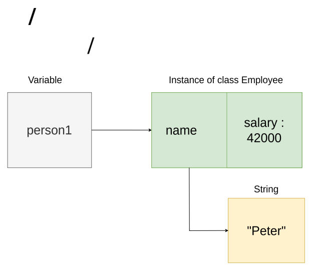
Like for class variables, we have to be careful with shadowing. Without `this.` in front of the variable name, the Java compiler will assume that you mean the parameter variable. It's a common mistake to write something like:

```
class Employee {
    String name;
    int salary;

    Employee(String name, int salary) {
        name = name;        //  oops, this.name is not changed here!
        salary = salary;    //  oops, this.salary is not changed here!
    }
}
```

2.1.3 Mental model

Like array variables and `String` variables, object variables contain a reference to the object in your computer's main memory. The object itself contains the instance variables. Note that an instance variable can be again a reference. For our employee `Peter`, we get the following structure:

Java code	In memory during execution

`Employee person1 =`
`new Employee("Peter", 42000);`
	 /
	/
	Variable
	person1
	Instance of class Employee
	name
	salary : 42000
	String
	"Peter"

Because of this, what we have already said about array variables and `String` variables also holds for object variables: assigning an object variable to another object variable only copies the reference. Comparing two object variables will only compare the references, not the content of the objects:

```
Employee person1 = new Employee("Peter", 42000);
Employee person2 = new Employee("Peter", 42000);
System.out.println(person1 == person2);      // false. Two different objects.

```

(continues on next page)

(continued from previous page)

```
Employee person3 = person1;
System.out.println(person1 == person3);      // true. Same object.
```

2.1.4 Working with objects

Many things that you can do with primitive types and strings, you can also do them with objects. For example, you can create arrays of objects. The elements of a new array of objects are automatically initialized to `null`, as shown in this example:

```
Employee[] myTeam = new Employee[3];
myTeam[0] = new Employee("Peter", 42000);
myTeam[1] = new Employee("Anna", 45000);
System.out.println(myTeam[0].name);       // is "Peter"
System.out.println(myTeam[1].name);       // is "Anna"
System.out.println(myTeam[2].name);       // Error! myTeam[2] is null
```

You can also have class variables and instance variables that are object variables. Again, they will be automatically initialized to `null`, if you don't provide an initial value. In the following example, we have added a new instance variable `boss` to our `Employee`:

```
class Employee {
    String name;
    int salary;
    Employee boss;

    Employee(String name, int salary, Employee boss) {
        this.name = name;
        this.salary = salary;
        this.boss = boss;
    }
}

public class Main {
    public static void main(String[] args) {
        // Anna has no boss
        Employee anna = new Employee("Anna", 45000, null);

        // Anna is the boss of Peter
        Employee peter = new Employee("Peter", 42000, anna);
    }
}
```

Exercise for you: Take a sheet of paper and draw the mental model graph for the object representing Peter.

Question: In the above example, what value do we give to the `boss` instance variable of an employee who has no boss?

2.1.5 Methods

In the following example, we define a static method `increaseSalary()` to increase the salary of an employee:

```java
class Employee {
    String name;
    int salary;

    Employee(String name, int salary) {
        this.name = name;
        this.salary = salary;
    }
}

public class Main {
    static void increaseSalary(Employee employee, int raise) {
        // we only raise the salary if the raise is less than 10000
        if (raise < 10000) {
            employee.salary += raise;
        }
    }

    public static void main(String[] args) {
        Employee anna = new Employee("Anna", 45000);
        Employee peter = new Employee("Peter", 45000);

        // Anna and Peter get a salary raise
        increaseSalary(anna, 2000);
        increaseSalary(peter, 3000);

        System.out.println("New salary of Anna is " + anna.salary);
        System.out.println("New salary of Peter is " + peter.salary);
    }
}
```

The above code works. But in Object-Oriented Programming (OOP) languages like Java, we generally prefer that all methods that modify instance variables of an object are put inside the class definition. In a large program, this makes it easier to understand who is doing what with an object. To implement this, we replace the static method `increaseSalary()` of the `Main` class by a non-static method in the `Employee` class:

```java
class Employee {
    String name;
    int salary;

    Employee(String name, int salary) {
        this.name = name;
        this.salary = salary;
    }

    void increaseSalary(int raise) {
        if (raise < 10000) {
            this.salary += raise;
        }
    }
```

(continues on next page)

(continued from previous page)

```java
}

public class Main {
    public static void main(String[] args) {
        Employee anna = new Employee("Anna", 45000);
        Employee peter = new Employee("Peter", 45000);

        // Anna and Peter get a salary raise
        anna.increaseSalary(2000);
        peter.increaseSalary(3000);

        System.out.println("New salary of Anna is "+ anna.salary);
        System.out.println("New salary of Peter is "+ peter.salary);
    }
}
```

Because `increaseSalary()` is now a non-static method of `Employee`, we can directly call it on an Employee object. No parameter `employee` is needed because, inside the method, the `this` keyword is a reference to the object on which the method has been called. Therefore, we just write `anna.increaseSalary(2000)` to change the salary of Anna.

2.1.6 Restricting access

The nice thing about our `increaseSalary()` method is that we can make sure that raises are limited to 10000 Euro :) However, nobody stops the programmer to ignore that method and manually change the salary:

```java
Employee anna = new Employee("Anna", 45000, null);
anna.salary += 1500000;   // ha!
```

This kind of mistake can quickly happen in a large program with hundreds of classes. We can prevent this by declaring the instance variable `salary` as `private`:

```java
class Employee {
    String name;
    private int salary;

    Employee(String name, int salary) {
        this.name = name;
        this.salary = salary;
    }

    void increaseSalary(int raise) {
        if (raise < 10000) {
            this.salary += raise;
        }
    }
}
```

A private instance variable is only accessible *inside* the class. So the access `anna.salary += 150000` in the `Main` class doesn't work anymore. Mission accomplished…

Unfortunately, that's a bit annoying because it also means that we cannot access anymore Anna's salary in `System.out.println("New salary of Anna is "+anna.salary)`. To fix this, we can add a method `getSalary()` whose only purpose is to give us the value of the private `salary` variable. Here is the new version of the code:

```java
class Employee {
    String name;
    private int salary;

    Employee(String name, int salary) {
        this.name = name;
        this.salary = salary;
    }

    void increaseSalary(int raise) {
        if (raise < 10000) {
            this.salary += raise;
        }
    }

    int getSalary() {
        return this.salary;
    }
}

public class Main {
    public static void main(String[] args) {
        Employee anna = new Employee("Anna", 45000);

        anna.increaseSalary(2000);

        System.out.println("New salary of Anna is "+ anna.getSalary());
    }
}
```

2.1.7 Inheritance

Let's say we are writing a computer game, for example an RPG (role-playing game). We implement weapons as objects of the class `Weapon`. The damage that a weapon can inflict depends on its level. The price of a weapon also depends on its level. The code could look like this:

```java
class Weapon {
    private int level;
    private String name;

    Weapon(String name, int level) {
        this.name = name;
        this.level = level;
    }

    int getPrice() {
        return this.level * 500;
    }

    int getSimpleDamage() {
        return this.level * 10;
```

(continues on next page)

(continued from previous page)

```
    }

    int getDoubleDamage() {
        return this.getSimpleDamage() * 2;
    }
}

public class Main {
    public static void main(String[] args) {
        Weapon weapon;

        weapon = new Weapon("Small dagger", 2);
        System.out.println("Price is " + weapon.getPrice());
        System.out.println("Simple damage is " + weapon.getSimpleDamage());
        System.out.println("Double damage is " + weapon.getDoubleDamage());
    }
}
```

Before you continue, carefully study the above program and make sure that you understand what it does. Run it in IntelliJ. Things are about to get a little more complicated in the following!

In our game, there is also a special weapon type, the *Mighty Swords*. These swords always deal a damage of 1000, independently of their level. In Java, we can implement this new weapon type like this:

```
class MightySword extends Weapon {
    MightySword(String name, int level) {
        super(name,level);
    }

    @Override
    int getSimpleDamage() {
        return 1000;
    }
}
```

According to the first line of this code, the class `MightySword` *extends* the class `Weapon`. We say that `MightySword` is *a subclass* (or *subtype*) of `Weapon`, or we can say that `Weapon` is a *superclass* of `MightySword`. In practice, this means that everything we can do with objects of the class `Weapon` we can also do with objects of the class `MightySword`:

```
public static void main(String[] args) {
    Weapon weapon;

    weapon = new MightySword("Magic sword", 3);
    System.out.println("Price is " + weapon.getPrice());
    System.out.println("Simple damage is " + weapon.getSimpleDamage());
    System.out.println("Double damage is " + weapon.getDoubleDamage());
}
```

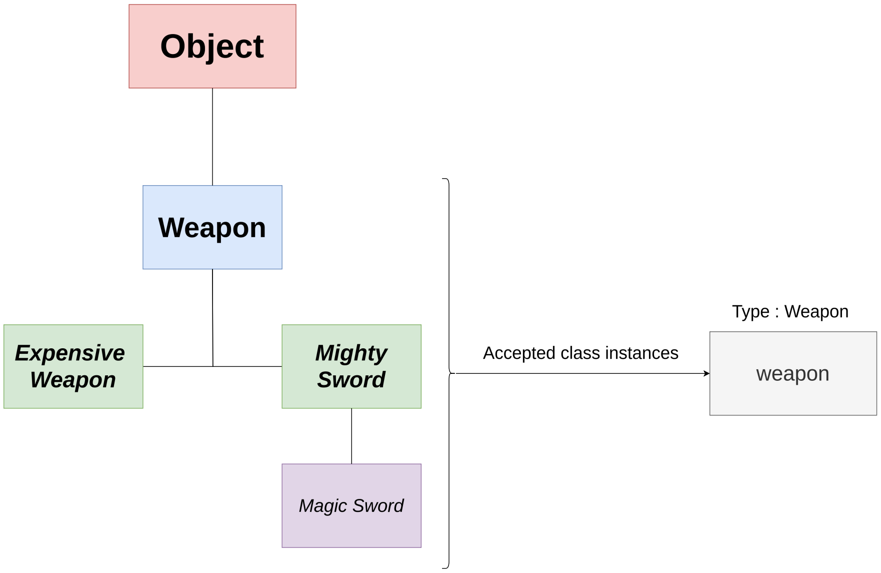
At first glance, there seems to be a mistake in the above `main()` method. Why is the line

```
weapon = new MightySword("Magic sword", 3);
```

not a type error? On the left, we have the variable `weapon` of type `Weapon` and on the right we have a new object of `MightySword`. But this is acceptable for the compiler because Java has the following rule:

Rule 1: A variable of type X can hold a reference to an object of class X or to an object of a subclass of X.

Because of rule 1, the compiler is perfectly happy with putting a reference to a `MightySword` object in a variable declared as type `Weapon`. For Java, MightySword instances are just special Weapon instances.



The next line of the `main()` method looks strange, too:

```
System.out.println("Price is " + weapon.getPrice());
```

Our class `MightySword` has not defined a method `getPrice` so why can we call `weapon.getPrice()`? This is another rule in Java:

Rule 2: The subclass inherits the methods of its superclass. Methods defined in a class X can be also used on objects of a subclass of X.

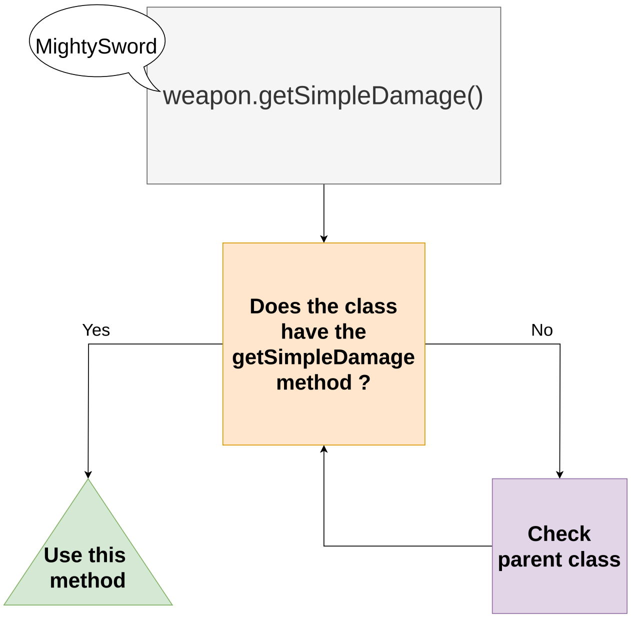
Let's look at the next line. It is:

```
System.out.println("Simple damage is " + weapon.getSimpleDamage());
```

Just by looking at this line and the line `Weapon weapon` at the beginning of the `main()` method, you might expect that `weapon.getSimpleDamage()` calls the `getSimpleDamage()` method of the `Weapon` class. However, if you check the output of the program, you will see that the method `getSimpleDamage()` of the class `MightySword` is called. Why? Because `weapon` contains a reference to a `MightySword` object. The rule is:

Rule 3: Let x be a variable of type X (where X is a class) and let's assign an object of class Y (where Y is a subclass of X) to x. When you call a method on x and the method is defined in X and in Y, the JVM will execute the method defined in Y.

For instances of the class `MightySword`, calling `getSimpleDamage()` will always execute the method defined in that class. We say that the method `getSimpleDamage()` in `MightySword` *overrides* the method definition in the class `Weapon`. For that reason, we have marked the method in `MightySword` with the so-called `@Override` annotation.



With the above three rules, can you guess what happens in the next line?

```
System.out.println("Double damage is " + weapon.getDoubleDamage());
```

According to rule 2, the class `MightySword` inherits the method `getDoubleDamage()` of the class `Weapon`. So, let's check how that method was defined in the class `Weapon`:

```
int getDoubleDamage() {
    return this.getSimpleDamage() * 2;
}
```

The method calls `this.getSimpleDamage()`. Which method `getSimpleDamage()` will be called? The one defined in `Weapon` or the one in `MightySword`? To answer this question, you have to remember rule 3! The `this` in `this.getSimpleDamage()` refers to the object on which the method was called. Since our method is an object of the class `MightySword`, the method `getSimpleDamage()` of `MightySword` will be called. The fact that `getDoubleDamage` is defined in the class `Weapon` does not change rule 3.

2.1.8 Super

There is one thing left in our `MightySword` class that we have not yet explained. It's the constructor:

```
class MightySword extends Weapon {

    MightySword(String name, int level) {
        super(name,level);
    }

    ...
 }
```

In the constructor, the keyword `super` stands for the constructor of the superclass of `MightySword`, that is `Weapon`. Therefore, the line `super(name,level)` simply calls the constructor as defined in `Weapon`.

super can also be used in methods. Imagine we want to define a new weapon type *Expensive Weapon* that costs exactly 100 more than a normal weapon. We can implement it as follows:

```java
class ExpensiveWeapon extends Weapon {

    ExpensiveWeapon(String name, int level) {
        super(name,level);
    }

    @Override
    int getPrice() {
        return super.getPrice() + 100;
    }
}
```

The expression super.getPrice() calls the method getPrice() as defined in the superclass Weapon. That means that the keyword super can be used to call methods of the superclass, which would normally not be possible for overridden methods because of rule 3.

2.1.9 The @Override annotation

The @Override annotation is not strictly necessary in Java (the compiler doesn't need it for itself), but it helps you to avoid mistakes. For example, imagine you made a spelling error when you wrote the name of getSimpleDamage():

```java
class MightySword extends Weapon {
    MightySword(String name, int level) {
        super(name,level);
    }

    @Override
    int getSimpleDamag() {  //  oops, we forgot the "e" in "getSimpleDamage()"
        return 1000;
    }
}
```

Because of your spelling error, the code above actually does not override anything. It just introduces a new method getSimpleDamag(). But thanks to the @Override annotation, the Java compiler can warn us that there is a problem.

2.1.10 Extending, extending,…

A subclass cannot only override methods of its superclass, it can also add new instance variables and new methods. For example, we can define a new type of Mighty Swords that can do magic damage:

```java
class MagicSword extends MightySword {
    private int magicLevel;

    MagicSword(String name, int level, int magicLevel) {
        super(name,level);  // call the constructor of MightySword
        this.magicLevel = magicLevel;
    }

    int getMagicDamage() {
```

(continues on next page)

(continued from previous page)

```java
        return this.magicLevel * 5;
    }
}
```

As you can see, you can create subclasses of subclasses. Note that the constructor uses again `super` to first call the constructor of the superclass and then initializes the new instance variable `magicLevel`.

How can we call the method `getMagicDamage()`? Can we do this:

```java
Weapon weapon = new MagicSword("Elven sword", 7, 3);
System.out.println(weapon.getMagicDamage());
```

The answer is no! Rule 3 is only applied to methods that are defined in the subclass *and* in the superclass. This is not the case for `getMagicDamage()`. In this situation, the Java compiler will not accept the call `weapon.getMagicDamage()` because, just by looking at the variable declaration `Weapon weapon`, it cannot tell that the object referenced by the variable `weapon` really has a method `getMagicDamage`. You might think that the compiler is a bit stupid here, but remember that this is just a simple example and the programmer could try to do some strange things that are difficult to see for the compiler:

```java
Weapon weapon = new MagicSword("Elven sword", 7, 3);
weapon = new Weapon("Dagger", 1);
System.out.println(weapon.getMagicDamage());  // does not compile, fortunately!
```

To be able to call `getMagicDamage()`, you have to convince the compiler that the variable contains a reference to a Magic Sword object. For example, you could change the type of the variable:

```java
MagicSword weapon = new MagicSword("Elven sword", 7, 3);
System.out.println(weapon.getMagicDamage());
```

In this way, it's 100% clear for the compiler that the variable definitely refers to a `MagicSword` object (or to an object of a subclass of `MagicSword`; remember rule 1).

Alternatively, you can do a type cast:

```java
Weapon weapon = new MagicSword("Elven sword", 7, 3);
System.out.println(((MagicSword) weapon).getMagicDamage());
```

However, be careful with type casts. The compiler will accept them but if you do a mistake, you will get an error during program execution:

```java
Weapon weapon = new Weapon("Dagger", 1);
System.out.println(((MagicSword) weapon).getMagicDamage());  // oh oh...
```

2.1.11 Polymorphism

The three rules make it possible to write code and data structures that can be used with objects of different classes. For example, thanks to rule 1, you can define an array that contains different types of weapons:

```java
Weapon[] inventory = new Weapon[3];
inventory[0] = new Weapon("Dagger", 2);
inventory[1] = new MagicSword("Elven sword", 7, 3);
inventory[2] = new ExpensiveWeapon("Golden pitchfork", 3);
```

And thanks to rule 2 and 3, you can write methods that work for different types of weapons:

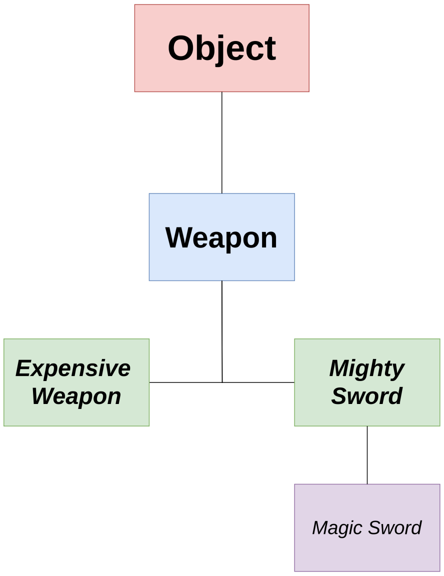
```
int getPriceOfInventory(Weapon[] inventory) {
    int sum = 0;
    for (Weapon weapon : inventory) {
        sum += weapon.getPrice();
    }
    return sum;
}
```

Although the above method `getPriceOfInventory()` looks like it is only meant for objects of class `Weapon`, it also works for all subclasses of `Weapon`. This is called *Subtype Polymorphism*. If you have for example an object of class `ExpensiveWeapon` in the array, rule 3 will guarantee that `weapon.getPrice()` will call the method defined in `ExpensiveWeapon`.

The conclusion is that there is a difference between what the compiler sees in the source code and what actually happens when the program is executed. When the compiler sees a method call like `weapon.getPrice()` in your source code, it only checks whether the method exists in the declared type of the variable. But during program execution, what is important is which object is actually referenced by the variable. We say that **type checking by the compiler is static**, but **method calls by the JVM are dynamic**.

2.1.12 The class hierarchy

If we take all the different weapon classes that we created in the previous examples, we get a so-called "class hierarchy" that shows the subclass-superclass relationships between them:



The class `Object` that is above our `Weapon` class was not defined by us. It is automatically created by Java and is the superclass of *all* non-primitive types in Java, even of arrays and strings! A variable of type `Object` therefore can refer to any non-primitive value:

```
Object o;
o = "Hello";                                   // okay
o = new int[]{1, 2, 3};                          // okay, too
o = new MagicSword("Elven sword", 7, 3);       // still okay!
```

The documentation of `Object` can be found at https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html. The class defines several interesting methods that can be used on all objects. One of them is the `toString()` method. This method is very useful because it is called by frequently used methods like `String.valueOf()` and `System.out.println()` when you call them with an object as parameter. Therefore, if we override this method in our own class, we will get a nice output:

```java
class Player {
    private String name;
    private int birthYear;

    Player(String name, int birthYear) {
        this.name = name;
        this.birthYear = birthYear;
    }

    @Override
    public String toString() {
        return "Player " + this.name + " born in " + this.birthYear;
    }
}

public class Main {
    public static void main(String[] args) {
        Player peter = new Player("Peter", 1993);
        System.out.println(peter);   // this will call toString() of Player
    }
}
```

The method `toString()` is declared as `public` in the class `Object` and, therefore, when we override it we have to declare it as public, too. We will talk about the meaning of `public` later.

Another interesting method defined by `Object` is `equals()`. We have already learned that we have to use the method `equals()` when we want to compare the content of two strings because the equality operator == only compares references. This is also recommended for your own objects. However, comparing objects is more difficult than comparing strings. For our class `Player` shown above, when are two players equal? The Java language cannot answer this question for us, so we have to provide our own implementation of `equals()`. For example, we could say that two `Player` objects are equal if they have the same name and the same birth year:

```java
import java.util.Objects;

class Player {
    private String name;
    private int birthYear;

    Player(String name, int birthYear) {
        this.name = name;
        this.birthYear = birthYear;
    }

    @Override
    public boolean equals(Object obj) {
        if (this == obj) {
            return true;    // same object!
        }
        else if (obj == null) {
            return false;   // null parameter
```

(continues on next page)

(continued from previous page)

```java
        }
        else if (this.getClass() != obj.getClass()) {
            return false;   // different types
        }
        else {
            Player p = (Player) obj;
            return p.name.equals(this.name) && p.birthYear == this.birthYear;
        }
    }

    @Override
    public int hashCode() {
        return Objects.hash(this.name, this.birthYear);
    }
}

public class Main {
    public static void main(String[] args) {
        Player peter1 = new Player("Peter", 1993);
        Player peter2 = new Player("Peter", 1993);
        System.out.println(peter1.equals(peter2));    // true
        System.out.println(peter1.equals("Hello"));   // false
        System.out.println(peter1.equals(null));      // false
    }
}
```

What's happening in the above code? One difficulty with `equals()` is that it can be called with a `null` argument or with an object that is not an instance of `Player`. So, before we can compare the name and the birth year of a `Player` object with another `Player` object, we first have to do some tests. One of them is whether the object on which `equals()` was called (`this`) and the other object (`obj`) have the same type:

```java
else if (this.getClass() != obj.getClass()) {
```

If all those tests pass we can finally compare the name and birth year of `this` and the other Player object.

Note that there are some other difficulties with `equals()` that we will not discuss here. They are related to the `hashCode()` method that you have to always override together with `equals()`, as shown above.

2.1.13 ArrayList

Using the class `Object` can be useful in situations where we want to write methods that work with all types of objects. For example, we have seen before that a disadvantage of arrays in Java over lists in Python is that arrays cannot change their size. In the package `java.util`, there is a class `ArrayList` that can do that:

```java
import java.util.ArrayList;

public class Main {
    public static void main(String[] args) {
        ArrayList list = new ArrayList();

        // add two elements to the end of the list
        list.add("Hello");
```

(continues on next page)

(continued from previous page)

```
        list.add(new int[]{1, 2, 3});

        System.out.println( list.size() );    // number of elements
        System.out.println( list.get(0) );    // first element
    }
}
```

As you can see in the above example, the method `add()` of `ArrayList` accepts any reference (including to arrays and strings) as argument. Very simplified, you can imagine that the `ArrayList` class looks like this:

```
public class ArrayList {
    // the added elements
    private Object[] elements;

    public void add(Object obj) {
        // this method adds "obj" to the array
        // ...
    }

    public Object get(int index) {
        // this method returns the object at position "index"
        // ...
    }
}
```

2.1.14 For loops on ArrayList

`for` loops also work on “ArrayList”:

```
ArrayList list = new ArrayList();
list.add("Hello");
list.add("World");

// simple for loop
for (Object obj : list) {
    System.out.println(obj);
}

// complex for loop
for (int i = 0; i < list.size(); i++) {
    System.out.println(list.get(i));
}
```

2.1.15 Boxing and unboxing

Unfortunately, primitive types are not subclasses of `Object`. Therefore, we cannot simple add an `int` value to an ArrayList, at least not without the help of the compiler:

```
list.add(3);  // does that work?
```

One way to solve this problem is to write a new class with the only purpose to store the `int` value in an object that we can then add to the list:

```
class IntObject {
    int value;

    IntObject(value) {
        this.value = value;
    }
}

public class Main {
    public static void main(String[] args) {
        ArrayList list = new ArrayList();

        list.add(new IntObject(3));
    }
}
```

This trick is called *boxing* because we put the primitive-type value 3 in a small “box” (the `IntObject` object). Fortunately, we actually don’t have to write our own class `IntObject`, because the `java.lang` package already contains a class that does exactly that:

```
// Integer is a class defined in the java.lang package
Integer value = Integer.valueOf(3);
list.add(value);
```

The `java.lang` package also contains equivalent classes `Long`, `Float`, etc. for the other primitive types.

Note that boxing is quite cumbersome and it is only needed in Java because primitive types are not subclasses of `Object`. However, we get a little bit of help from the compiler. In fact, the compiler can do the boxing for you. This is called **autoboxing**. You can just write:

```
list.add(3);  // this automatically calls "Integer.valueOf(3)"
```

Autoboxing is not limited to the `ArrayList` class. It works for all situations where you assign a primitive-type value to a variable that has a matching class type. The opposite direction, unboxing, is also done automatically by the compiler:

```
// autoboxing
// this is identical to:
//    Integer value = Integer.valueOf(3);
Integer value = 3;

// auto-unboxing
// this is identical to:
//    int i = value.intValue();
int i = value;
```

2.1.16 ArrayList and Generics

The way `ArrayList` uses `Object` to be able to store all kinds of objects has a big disadvantage. Since the `get` method has the return type `Object`, we have to do a type cast if we want again the original type of the object that we added to the list:

```java
ArrayList list = new ArrayList();

list.add("Hello");
list.add("World");

int len = ((String) list.get(0)).length();
```

Although *we* know that the list only contains strings, the compiler needs the type cast before we can call the method `length()`. This is not only cumbersome, but can also lead to errors that only appear when the program is executed.

Fortunately, Java has a feature called *Generics* that allows us to simplify the above code as follows:

```java
ArrayList<String> list = new ArrayList<String>();

list.add("Hello");
list.add("World");

int len = list.get(0).length();
```

The syntax `ArrayList<String>` tells the compiler that the `add()` method of our list will only accept `String` objects as arguments and that the `get()` method will only return `String` objects. In that way, the type cast is not needed anymore (actually, the type cast is still done but you don't see it because the compiler automatically adds it in the class file).

You will see more examples of *Generics* later in this book. To give you a first taste, let's see what the `ArrayList` class looks like in reality:

```java
public class ArrayList<E> {    // type parameter E
    private Object[] elements;

    public void add(E obj) {
        // ...
    }

    public E get(int index) {
        // ...
    }
}
```

The `E` that you can see in the first line and in the method definitions is a *type parameter*. It represents the type of the element that we want to store in the list. By creating our list with

```java
ArrayList<String> list = new ArrayList<String>();
```

we are telling the compiler that it should assume that `E = String`, and accordingly the methods `add()` and `get()` will be understood as `void add(String obj)` and `String get(int index)`.

2.1.17 Method overloading with different parameters

In Java, it is allowed to have two methods with the same name as long as they have different parameters. This is called *method overloading*. Here is an example:

```java
class Player {
    private String name;
    private int birthYear;

    Player(String name, int birthYear) {
        this.name = name;
        this.birthYear = birthYear;
    }

    public void set(String name) {
        this.name = name;
    }

    public void set(String name, int birthYear) {
        this.name = name;
        this.birthYear = birthYear;
    }
}
```

If we call the `set()` method, the Java compiler knows which of the two methods you wanted to call by looking at the parameters:

```java
Person person = new Person("Peter", 1993);
person.set("Pierre", 1993);     // this is the set method with parameters String and int
```

2.1.18 Overloading with subclass parameters

You have to be careful when you write overloaded methods where the parameters are classes and subclasses. Here is a minimal example of a `Player` class with such an overloaded method:

```java
class Weapon {
    // ...
}

class MightySword extends Weapon {
    // ...
}

class Player {
    Weapon weapon;
    int power;

    void giveWeapon(Weapon weapon) {
        this.weapon = weapon;
        this.power = 0;
    }

```

(continues on next page)

(continued from previous page)

```
    void giveWeapon(MightySword weapon) {
        this.weapon = weapon;
        this.power = 10;   // a Mighty Sword increases the power of the player
    }
}

public class Main {
    public static void main(String[] args) {
        Player player = new Player();

        Weapon weapon = new MightySword();
        player.giveWeapon(weapon);

        System.out.println(player.power);
    }
}
```

What will `System.out.println(player.power)` print after we gave a Mighty Sword to the player?

Surprisingly, it will print `0`. The method `void giveWeapon(MightySword weapon)` is **not** called although we called `giveWeapon()` with a `MightySword` object! The explanation for this is that the Java compiler only looks at the type of the variable *as declared in the source code* when deciding which method to call. In our example, the type of the variable `weapon` is `Weapon`, therefore the method `void giveWeapon(Weapon weapon)` is called. The compiler cannot know that the variable will contain a reference to a `MightySword` object during program execution.

Lesson learned: **Method calls in Java are only dynamically decided for the object on which the method is called (remember rule 3!). They are not dynamic for the arguments of the method.**

The correct way to call `giveWeapon()` for Mighty Swords is:

```
MightySword weapon = new MightySword();
player.giveWeapon(weapon);
```

or just:

```
player.giveWeapon(new MightySword());
```

2.1.19 Overloading with closest match

What happens if we call an overloaded method but there is no version of the method that exactly matches the type of the argument? Here is the same example as above, but with a third class `MagicSword` that is a subclass of `MightySword`:

```
class Weapon {
    // ...
}

class MightySword extends Weapon {
    // ...
}

class MagicSword extends MightySword {
    // ...
}
```

(continues on next page)

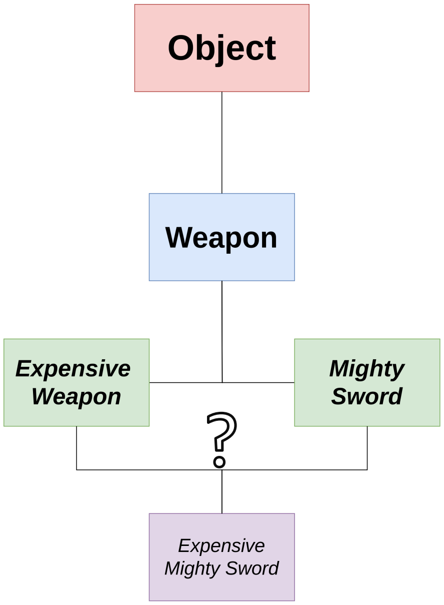
(continued from previous page)

```java
class Player {
    Weapon weapon;
    int power;

    void giveWeapon(Weapon weapon) {
        this.weapon = weapon;
        this.power = 0;
    }

    void giveWeapon(MightySword weapon) {
        this.weapon = weapon;
        this.power = 10;
    }
}

public class Main {
    public static void main(String[] args) {
        Player player = new Player();

        player.giveWeapon(new MagicSword());

        System.out.println(player.power);
    }
}
```

Which one of the two `giveWeapon()` will be called if the argument is a `MagicSword` object? In this situation, the compiler will choose the method with the closest type to `MagicSword`, that is `void giveWeapon(MightySword weapon)`.

2.1.20 Multiple Inheritance

If we look back at our examples with the Weapon subclasses `ExpensiveWeapon` and `MightySword`, we might be tempted to create a new class `ExpensiveMightySword` that inherits from both subclasses:



Unfortunately, inheriting from two (or more) classes is **not allowed** in Java. The reason for this is the *diamond problem* that occurs when a class inherits from two classes that are subclasses of the same class (the problem is named after the diamond shape of the resulting class hierarchy). The following illegal Java program illustrates the problem:

```
class Weapon {
    int level;

    int getPrice() {
        return 100;
    }
}

class ExpensiveWeapon extends Weapon {
    @Override
    int getPrice() {
        return 1000;
    }
}

class MightySword extends Weapon {
    @Override
    int getPrice() {
        return 500 * level;
    }
}

// Not allowed in Java!
// You cannot extend TWO classes.
class ExpensiveMightySword extends ExpensiveWeapon, MightySword {
}

public class Main {
    public static void main(String[] args) {
        Weapon weapon = new ExpensiveMightySword();
```

(continues on next page)

(continued from previous page)

```
        System.out.println(weapon.getPrice());        // ???
    }
}
```

Which getPrice implementation should be called in the println() statement? The one from ExpensiveWeapon or the one from MightySword? Because it is not clear in this situation what the programmer wanted, multiple inheritance is forbidden in Java. Other programming languages allow multiple inheritance under specific circumstances, or have additional rules to decide which method to call. For example, the C# language would require for our example that the ExpensiveMightySword class overrides the getPrice() method. In Python, the getPrice() method of the ExpensiveWeapon class would be called because that class appears first in the line

```
class ExpensiveMightySword extends ExpensiveWeapon, MightySword {
```

If you want to know more about how other programming languages handle multiple inheritance and the diamond problem, you can check https://en.wikipedia.org/wiki/Multiple_inheritance.

However, Java has another concept, the interface, which can be used as a substitute for multiple inheritance in many situations. You will learn more about interfaces later.

2.1.21 The final keyword

Like the private keyword, the final keyword does not change the behavior of your program. Its job is to prevent you from making mistakes in your code (you will later see other situations where the final keyword is important).

Its meaning depends on where you use it.

2.1.22 Final parameter variables

If you declare a parameter variable as final, its value cannot be changed inside the method. This prevents accidents like the following:

```
 // calculate the sum of the numbers 1 to n
int calculateSum ( final int n){   // <--- did you see the "final" ?
    int sum = 0;
    for (int i = 1; i < n; i++) {
        n += i;      // oops, I wanted to write sum += i
    }
    return sum;
}
```

In the above example, the statement n+=i will not be accepted by the compiler because the parameter n was declared as final.

Note that if a variable contains a reference to an array or an object, declaring it as final does not prevent the contents of the array or object from being changed. This is also true for the other usages of final explained in the next sections. Here is an example:

```
void increment(final int[] a) {
    a[0]++;         // this still works
}
```

2.1.23 Final local variables

Local variables declared as `final` cannot change their value after they have been initialized. The following code will not be accepted by the compiler:

```
int calculateSumSquare(int n) {
    final int n2 = n * n;       // <--- did you see the "final" ?
    int sum = 0;
    for (int i = 1; i < n2; i++) {
        n2 += i;      // oops, I wanted to write sum += i
    }
    return sum;
}
```

2.1.24 Final methods

Methods declared as `final` cannot be overridden in a subclass. Declaring a method as `final` is useful in situations where you think that the method contains important code and you fear that a subclass could break the class by overriding it. The following code will not be accepted by the compiler:

```
class Person {
    String name, firstname;

    final String getFullName() {
        return firstname + " " + name;
    }
}

class Employee extends Person {
    @Override
    String getFullName() {      // not allowed. Method is "final" in "Person" class
        return "Wolverine";
    }
}
```

However, you should think carefully about whether you should declare a method as `final`, as this would drastically limit the flexibility of subclasses.

2.1.25 Final classes

Classes declared as `final` cannot be subclassed. The motivation to do this is similar to `final` methods. For example, the `String` class is final because all Java programs rely on its specific behavior as described in the documentation. Creating a subclass of it would cause a lot of problems.

2.1.26 Final class variables

Like `final` local variables, class variables declared as `final` cannot be changed after initialization. A typical use case is the declaration of a constant. Here is an example:

```java
class Physics {
    static final double SPEED_OF_LIGHT = 299792458; //  meters per second
}
```

The naming convention in Java recommends writing the names of constants in capital letters.

2.1.27 Final instance variables

Instance variables declared as `final` cannot be changed after initialization. However, unlike class variables, you will usually initialize them in the constructor. The following code demonstrates this:

```java
class Person {
    final String socialSecurityNumber;

    Person(String ssn) {
        this.socialSecurityNumber = ssn;
    }
}

public class Main {
    public static void main(String[] args) {
        Person person = new Person("123-456-789");
        person.socialSecurityNumber = "12";        // error!
    }
}
```

An important reason to declare an instance variable as `final` is when it is part of the “identity” of an object, i.e., something that should never change once the object has been created.

Note that a variable that cannot be modified after initialization can be also achieved without declaring it as `final`. In the above example, we could implement the immutable social security number also like this:

```java
class Person {
    private String socialSecurityNumber;

    Person(String ssn) {
        this.socialSecurityNumber = ssn;
    }

    final String getSSN() {  // "final" prevents overriding
        return this.socialSecurityNumber;
```

(continues on next page)

(continued from previous page)

```
    }
}
```

2.1.28 Packages

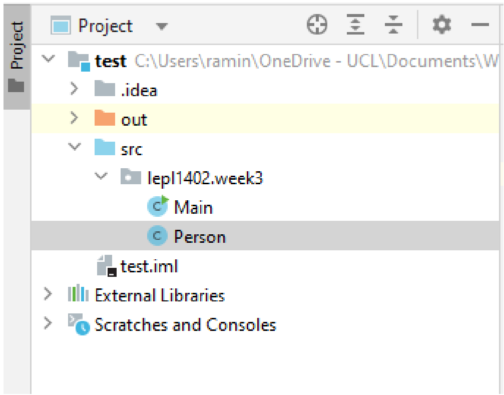
In all our small examples so far, we have put all classes in one single `.java` file. This is not very practical in larger projects consisting of dozens or hundreds of classes.

The general rule (or recommendation) in Java is that you should put each class in a separate .java file with the same name as the class.

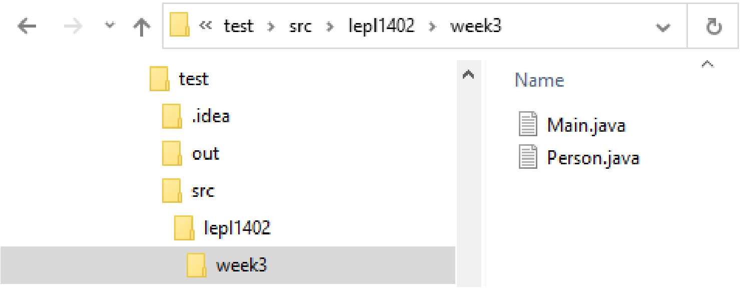
In addition, Java allows you to group classes into *packages* by writing a package statement in the first line of your `.java` file. For example, the following two `.java` files define two classes that are in the package `lepl1402.week3`:

```
// **********************************
// ****     File Person.java     ****
// **********************************

package lepl1402.week3;

class Person {
    final String socialSecurityNumber;

    Person(String ssn) {
        this.socialSecurityNumber = ssn;
    }
}

// **********************************
// ****      File Main.java      ****
// **********************************

package lepl1402.week3;

public class Main {
    public static void main(String[] args) {
        Person person = new Person("123-456-789");
    }
}
```

If you put your classes into packages, the Java compiler expects that you organize the source code files in your project in a directory structure that corresponds to the package names. In our example with the package `lepl1402.week3`, the .java files **must** be put in a directory `week3` inside a directory `lepl1402` in the `src` directory of your project. Here is what IntelliJ shows for the above project:



And here is how the directory structure of the project looks like in the file browser of Microsoft Windows:



If you do not write a package statement in your .java file (that's what we always did so far in our examples), the compiler puts your classes in the *unnamed package*. In that case, you don't need a special directory structure and you can put all your files directly into the src directory.

2.1.29 How to use multiple packages

In Java, packages are independent of each other. Classes that are in the same package can be used together, as shown in the example in the previous section with the Person class and the Main class.

However, classes that are in *different* packages do not "see" each other by default. For example, if we put the class Person into the package lepl1402.week3.example and we keep the class Main in the package lepl402.week3, we have to change our code:

```
// **********************************
// ****     File Person.java     ****
// **********************************

package lepl1402.week3.example;

public class Person {
    final String socialSecurityNumber;

    public Person(String ssn) {
        this.socialSecurityNumber = ssn;
    }
}

// **********************************
// ****      File Main.java      ****
```

(continues on next page)

(continued from previous page)

```java
// **********************************

package lepl1402.week3;

import lepl1402.week3.example.Person;

public class Main {
    public static void main(String[] args) {
        Person person = new Person("123-456-789");
    }
}
```

In our example, we have made three modifications:

1. We have declared the class `Person` as `public`. Only classes that are public can be used by classes in other packages! If a class is not declared as public, it can only be used by classes of the same package.
2. We have declared the constructor method of `Person` as `public`. Again, only public methods can be used by classes in other packages.
3. We have added an `import` statement to our file `Main.java` file. This statement tells the compiler (and the JVM) in which package the class `Person` is located that the `Main class` wants to use. The identifier `lepl1402.week3.example.Person` is called the *fully qualified name* of the class `Person`.

As an alternative to the `import` statement, you could directly use the fully qualified name of the `Person` class in the main method, but this makes the code a bit harder to read:

```java
// **********************************
// ****      File Main.java      ****
// **********************************

package lepl1402.week3;

public class Main {
    public static void main(String[] args) {
        lepl1402.week3.example.Person person
              = new lepl1402.week3.example.Person("123-456-789");
    }
}
```

2.1.30 Why are packages useful?

Packages have two advantages. First of all, with the `public` keyword, you can control for each class and each method in your package whether it can be used by classes in other packages. For example, we have already talked several times about the `java.lang` package that contains useful classes such as `String` or `Integer`. Those classes are declared as public, so everybody can use them. However, that package also contains classes like `CharacterData0E` that are only used internally by some classes in `java.lang` and that are therefore *not* declared as public.

The second advantage of packages is that they provide separate *namespaces*. This means that a package X and a package Y can both contain a class named `ABC`. By using the fully classified names (or an `import` statement), we can exactly tell the compiler whether we want to use class `X.ABC` or class `Y.ABC`. This becomes important when you write larger applications and you want to use packages written by other people. Thanks to the different packages, you don't have to worry about classes with identical names.

2.1.31 Access control

First, let's summarize what we have learned about the visibility of classes in packages:

1. Classes that are declared as `public` are visible in all packages.
2. Non-public classes are only visible inside their own package.

For class members (i.e., static and non-static methods, class variables, and instance variables), the rules are more complicated:

1. Members that are declared as `public` are accessible from all packages.
2. Members that are declared as `private` are only accessible inside their class.
3. Members that are declared as `protected` are only accessible inside their class and in subclasses of that class.
4. Members that have no special declaration are accessible inside the class and by all classes in the same package.

2.2 Abstract classes

An abstract class in Java is a class that cannot be instantiated on its own and is intended to be a parent class. Abstract classes are used when you want to provide a common base for different subclasses but do not want this base class to be instantiated on its own. They can contain both fully implemented (concrete) methods and abstract methods (methods without a body).

Imagine we are designing a geometric drawing program that incorporates scientific computations, such as calculating the area of various shapes. In this program, the formula for computing the area will be dependent on the specific shape, but there will also be common functionalities. For instance, each shape should have the capability to print information about itself. Additionally, the program is designed to allow users to define their own shapes.

Our design objective is to adhere to the crucial "Open/Closed Principle" (OCP) of object-oriented programming. This principle advocates that software entities (such as classes, modules, and functions) should be open for extension but closed for modification. This approach ensures that our program can grow and adapt over time without necessitating alterations to the existing, stable parts of the code.

Abstract classes become immensely valuable in this context. We can encapsulate all the common functionalities for handling the various geometric shapes into an abstract class, thereby avoiding code duplication. This abstract class will define methods that are common across all shapes, such as a method to print information about the shape. However, for specific functionalities that vary from one shape to another, such as the computation of area, we leave the method abstract.

```java
public abstract class Shape {
    protected String shapeName; // Instance variable to hold the name of the shape

    public Shape(String name) {
        this.shapeName = name;
    }

    // Abstract method to calculate the area of the shape
    public abstract double calculateArea();

    // A concrete method implemented in the abstract class
    public void displayShapeInfo() {
        System.out.println("The " + shapeName + " has an area of: " + calculateArea());
    }
}
```

With this design, introducing new shapes into the program is straightforward and does not require to change the structure of existing code. We simply add new subclasses for the new shapes.

```java
public class Circle extends Shape {
    private double radius;

    public Circle(double radius) {
        super("Circle");
        this.radius = radius;
    }

    @Override
    public double calculateArea() {
        return Math.PI * radius * radius;
    }
}

public class Rectangle extends Shape {
    private double length;
    private double width;

    public Rectangle(double length, double width) {
        super("Rectangle");
        this.length = length;
        this.width = width;
    }

    @Override
    public double calculateArea() {
        return length * width;
    }
}

public class Triangle extends Shape {
    private double base;
    private double height;

    public Triangle(double base, double height) {
        super("Triangle");
        this.base = base;
        this.height = height;
    }

    @Override
    public double calculateArea() {
        return 0.5 * base * height;
    }
}
```

To compute the total area of all shapes in an array, we can create a static method that takes an array of `Shape` objects as its parameter. This method will iterate on it, invoking the `calculateArea()` method on each `Shape` object, and accumulate the total area. This static method remains valid even if you introduce later a new shape in your library.

```java
class ShapeUtils {

    // Static method to compute the total area of an array of shapes
    public static double calculateTotalArea(Shape[] shapes) {
        double totalArea = 0.0;

        for (Shape shape : shapes) {
            totalArea += shape.calculateArea();
        }

        return totalArea;
    }

    public static void main(String[] args) {
        Shape[] shapes = {new Circle(5), new Rectangle(4, 5), new Triangle(3, 4)};
        double totalArea = calculateTotalArea(shapes);
        System.out.println("Total Area: " + totalArea);
    }
}
```

2.3 Interfaces

An interface in Java is a class that is completely abstract. In other words, none of its methods has a concrete implementation. Interfaces are used to group related methods with empty bodies. Interfaces specify what a class must do, but not how it does it.

One advantage of interfaces over abstract classes is the ability of a class to implement multiple interfaces. Remember that Java doesn't allow to *extend multiple classes*.

Therefore interfaces promote a higher degree of flexibility and modularity in software design than abstract classes, but they don't often the same facility in terms of factorization of the code.

```java
public interface Camera {
    void takePhoto();
    void recordVideo();
}

public interface MediaPlayer {
    void playAudio();
    void playVideo();
}
```

```java
public class Smartphone implements Camera, MediaPlayer {

    @Override
    public void takePhoto() {
        System.out.println("Taking a photo");
    }

    @Override
    public void recordVideo() {
```

(continues on next page)

(continued from previous page)

```
        System.out.println("Recording video");
    }

    @Override
    public void playAudio() {
        System.out.println("Playing audio");
    }

    @Override
    public void playVideo() {
        System.out.println("Playing video");
    }
}
```

2.4 Delegation

Let us consider the Book class below:

```
public class Book {
    private String title;
    private String author;
    private int publicationYear;

    public Book(String title, String author, int year) {
        this.title = title;
        this.author = author;
        this.publicationYear = year;
    }

    // ... getters, setters, and other methods ...
}
```

We aim to sort a collection of Book objects based on their titles in lexicographic order. This can be done by implementing the Comparable interface that requires to define the compareTo() method. The compareTo() method, when implemented within the Book class, leverages the inherent compareTo() method of the String class.

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

    public class Book implements Comparable<Book> {
        final String title;
        final String author;
        final int publicationYear;

        public Book(String title, String author, int year) {
            this.title = title;
            this.author = author;
            this.publicationYear = year;
        }
```

(continues on next page)

(continued from previous page)

```
        @Override
        public int compareTo(Book other) {
            return this.title.compareTo(other.title);
        }

        public static void main(String[] args) {
            List<Book> books = new ArrayList<>();
            books.add(new Book("The Great Gatsby", "F. Scott Fitzgerald", 1925));
            books.add(new Book("Moby Dick", "Herman Melville", 1851));
            books.add(new Book("1984", "George Orwell", 1949));

            Collections.sort(books);  // Sorts by title due to the implemented Comparable

            for (Book book : books) {
                System.out.println(book.getTitle());
            }
        }
    }
```

Imagine that the books are displayed on a website, allowing visitors to browse through an extensive catalog. To enhance user experience, the website provides a feature to sort the books not just by their titles, but also by other attributes: the author's name or the publication year.

Now, the challenge arises: Our current Book class design uses the Comparable interface to determine the natural ordering of books based solely on their titles. While this design works perfectly for sorting by title, it becomes restrictive when we want to provide multiple sorting criteria (for instance, sorting by author or publication year). Since the Comparable interface mandates a single compareTo() method, it implies that there's only one "natural" way to sort the objects of a class. This design decision binds us to sorting by title and makes it less straightforward to introduce additional sorting methods for other attributes.

A general important principle of object-oriented design is the *Open/Closed Principle (OCP)*: A software module (like a class or method) should be open for extension but closed for modification:

1. Open for Extension: This means that the behavior of the module can be extended or changed as the requirements of the application evolve or new functionalities are introduced.
2. Closed for Modification: Once the module is developed, it should not be modified to add new behavior or features. Any new functionality should be added by extending the module, not by making modifications to the existing code.

The so-called *Delegate Design Pattern* can help us improve our design and is a nice example of the OCP. In the example of Book, delegation occurs when the sorting algorithm (within Collections.sort()) calls the compare() method of the provided Comparator object. The responsibility of defining how two Book objects compare is delegated to the Comparator object, allowing for flexibility in sorting criteria without modifying the Book class or the sorting algorithm itself.

This delegation approach with Comparator has a clear advantage over inheritance because you can define countless sorting criteria without needing to modify or subclass the original Book class.

Here are the three Comparator classes, one for each sorting criterion:

```
import java.util.Comparator;

public class TitleComparator implements Comparator<Book> {
    @Override
```

(continues on next page)

(continued from previous page)

```java
    public int compare(Book b1, Book b2) {
        return b1.getTitle().compareTo(b2.getTitle());
    }
}

public class AuthorComparator implements Comparator<Book> {
    @Override
    public int compare(Book b1, Book b2) {
        return b1.getAuthor().compareTo(b2.getAuthor());
    }
}

public class YearComparator implements Comparator<Book> {
    @Override
    public int compare(Book b1, Book b2) {
        return Integer.compare(b1.getPublicationYear(), b2.getPublicationYear());
    }
}
```

As next example shows, we can now sort by title, author or publication year by just providing the corresponding comparator to the sorting algorithm.

```java
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

public class Main {
    public static void main(String[] args) {
        List<Book> books = new ArrayList<>();
        books.add(new Book("The Great Gatsby", "F. Scott Fitzgerald", 1925));
        books.add(new Book("Moby Dick", "Herman Melville", 1851));
        books.add(new Book("1984", "George Orwell", 1949));

        Collections.sort(books, new TitleComparator());  // Sort by title
        Collections.sort(books, new AuthorComparator()); // Sort by author
        Collections.sort(books, new YearComparator());   // Sort by publication year
    }
}
```

Exercise

You are developing a document management system. As part of the system, you have a `Document` class that contains content. You want to provide a printing capability for the `Document`.

Instead of embedding the printing logic directly within the `Document` class, you decide to use the delegate design pattern. This will allow the `Document` class to delegate the responsibility of printing to another class, thus adhering to the single responsibility principle.

Complete the code below.

```java
// The Printer interface
interface Printer {
    void print(String content);
```

(continues on next page)

(continued from previous page)

```java
}

// TODO: Implement the Printer interface for InkjetPrinter
class InkjetPrinter ... {
    ...
}

// TODO: Implement the Printer interface for LaserPrinter
class LaserPrinter ... {
    ...
}

// Document class
class Document {
    private String content;
    private Printer printerDelegate;

    public Document(String content) {
        this.content = content;
    }

    // TODO: Set the printer delegate
    public void setPrinterDelegate(...) {
        ...
    }

    // TODO: Print the document using the delegate
    public void printDocument() {
        ...
    }
}

// Demo
public class DelegateDemo {
    public static void main(String[] args) {
        Document doc = new Document("This is a sample document content.");

        // TODO: Set the delegate to InkjetPrinter and print
        ...

        // TODO: Set the delegate to LaserPrinter and print
        ...
    }
}
```

2.5 Observer

In computer science, it is considered as a good practice to have a loose coupling between objects (the opposite is generally referred to as a “spaghetti code”). Loose coupling allows for more modular and maintainable code.

The *Observer Design Pattern* is a pattern that we can use to have a loose coupling between objects.

We will first show how to use observers in the context of GUI development (Graphical User Interface), then will show how to implement observers.

2.5.1 Observer pattern on GUI components

In Java, the swing and awt packages facilitate the creation of Graphical User Interfaces (GUIs). Swing in Java uses a system based on the observer pattern to handle events, such as mouse clicks.

On the next example we have a solitary button that, when clicked, responds with the message “Thank you” to the user.

```java
    import javax.swing.JButton;
    import javax.swing.JFrame;
    import javax.swing.JOptionPane;
    import java.awt.event.ActionEvent;
    import java.awt.event.ActionListener;

    class ButtonActionListener implements ActionListener {
        @Override
        public void actionPerformed(ActionEvent e) {
            JOptionPane.showMessageDialog(null,"Thank you!");
        }
    }

    public class AppWithActionListener {
    public static void main(String[] args) {
        JFrame frame = new JFrame("Hello");
        frame.setSize(400, 200);
        frame.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);

        JButton button = new JButton("Press me!");
        button.addActionListener(new ButtonActionListener());
        frame.add(button);

        frame.setVisible(true);
    }
}
```

The ActionListener is an interface within Java that contains a single method: actionPerformed(). In our application, this interface is implemented by the ButtonActionListener concrete class. When invoked, it displays a dialog with the message “Thank you!” to the user. However, this setup remains inactive until we associate an instance of our ButtonActionListener to a button using the addActionListener() method. This ensures that every time the button is pressed, the actionPerformed() method of our listener gets triggered.

It is worth noting that the inner workings of how the button manages this relationship or stores the listener are abstracted away. What is crucial for developers to understand is the contract: The listener’s method will be invoked whenever the button is clicked. This process is often referred to as “attaching a callback” to the button, or as “registering an event handler” to the button. This concept echoes a well-known programming principle sometimes dubbed the Hollywood principle: “Don’t call us, we will call you.”

Although we have registered only one listener to the button, this is not a limitation. Buttons can accommodate multiple listeners. For example, a second listener could be added to track the total number of times the button has been clicked.

This setup exemplifies the observer design pattern from the perspective of end users, using the JButton as an illustration. Let's now delve into how to implement this pattern for custom classes.

2.5.2 Implementing the Observer pattern

Imagine a scenario where there's a bank account that multiple people, say family members, can deposit into. Each family member possesses a smartphone and wishes to be alerted whenever a deposit occurs. For the sake of simplicity, these notifications will be printed to the console. The complete source code is given next.

```java
public interface AccountObserver {
    public void accountHasChanged(int newValue);
}


class MyObserver implements AccountObserver {
    @Override
    public void accountHasChanged(int newValue) {
        System.out.println("The account has changed. New value: "+newValue);
    }
}

public class ObservableAccount {
    private int value ;
    private List<AccountObserver> observers = new LinkedList();

    public void deposit(int d) {
        value += d;
        for (AccountObserver o: observers) {
            o.accountHasChanged(value);
        }
    }

    public void addObserver(AccountObserver o) {
        observers.add(o);
    }

    public static void main(String [] args) {
        ObservableAccount account = new ObservableAccount();
        MyObserver observerFather = new MyObserver();
        MyObserver observerMother = new MyObserver();
        MyObserver observerGirl = new MyObserver();
        MyObserver observerBoy = new MyObserver();

        account.addObserver(observerFather);
        account.addObserver(observerMother);
        account.addObserver(observerGirl);
        account.addObserver(observerBoy);

        account.deposit(100); // prints 4X "The account has changed. New Value: 100"
        account.deposit(50);  // prints 4X "The account has changed. New Value: 150"
```

(continues on next page)

(continued from previous page)

```
    }
}
```

In this context, our bank account is the subject being observed. In our code, this will be modeled by the `ObservableAccount` class. This account maintains a balance, which can be incremented through a deposit function.

We require a mechanism to register observers (note: the wordings “observer” and “listener” are synonyms that can be used interchangeably) who wish to be informed about deposits. The `LinkedList` data structure is an excellent choice for this purpose: It offers constant-time addition and seamlessly supports iterators since it implements the `Iterable` interface. To add an `AccountObserver`, one would simply append it to this list. We have chosen not to check for duplicate observers in the list, believing that ensuring uniqueness is the user’s responsibility.

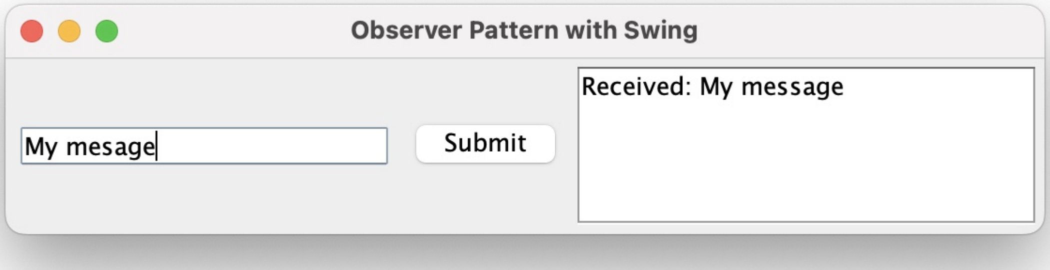
Whenever a deposit occurs, the account balance is updated, and subsequently, each registered observer is notified by invoking its `accountHasChangedMethod()`, which shares the updated balance.

It is important to note that in this specific implementation, the order of notification is determined by the sequence of registrations because we are using a list. However, from a user’s standpoint, the caller should never make the hypothesis that this order is always used. Indeed, one could replace the `LinkedList` by another collection, for instance a set, which would not guarantee the same order while iterating over the observers.

Exercise

In this exercise, you will use the Observer pattern in conjunction with the Java Swing framework. The application `MessageApp` provides a simple GUI (Graphical User Interface) where users can type a message and submit it. This message, once submitted, goes through a spell checker and then is meant to be displayed to observers.

Your task is to make it work as expected: when a message is submitted, it is corrected by the spell checker and it is appended in the text area of the app (use `textArea.append(String text)`).



My mesage

Received: My message

It is imperative that your design allows for seamless swapping of the spell checker without necessitating changes to the `MessageApp` class. Additionally, the `MessageSubject` class should remain decoupled from the `MessageApp`. It must not depend on it and should not even be aware that it exists.

Use the observer pattern in your design. You’ll have to add instance variables and additional arguments to some existing constructors. When possible, always prefer to depend on interfaces rather than on concrete classes when declaring your parameters. With the advances of Deep Learning, we anticipate that we will soon have to replace the existing `StupidSpellChecker` by a more advanced one. Make this planned change as simple as possible, without having to change your classes.

```java
import javax.swing.*;
import java.awt.event.*;

import java.util.ArrayList;
import java.util.List;


public class MessageApp {
    private JFrame frame;
    private JTextField textField;
    private JTextArea textArea;
    private JButton submitButton;

    public MessageApp() {

        frame = new JFrame("Observer Pattern with Swing");
        textField = new JTextField(16);
        textArea = new JTextArea(5, 20);
        submitButton = new JButton("Submit");

        frame.setLayout(new java.awt.FlowLayout());

        frame.add(textField);
        frame.add(submitButton);
        frame.add(new JScrollPane(textArea));

        // Hint: add an actionListner to the submitButon
        // Hint: use textField.getText() to retrieve the text

        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.pack();
        frame.setVisible(true);
    }

    public static void main(String[] args) {
        SwingUtilities.invokeLater(new Runnable() {
            public void run() {
                new MessageApp();
            }
        });
    }
}

interface SpellChecker {
        String correct(String sentence);
}

class StupidSpellChecker implements SpellChecker {
        public String correct(String sentence) {
        return sentence;
        }
}
```

(continues on next page)

(continued from previous page)

```java
interface MessageObserver {
    void updateMessage(String message);
}


class MessageSubject {

    private List<MessageObserver> observers = new ArrayList<>();
    private String message;

    public void addObserver(MessageObserver observer) {
        observers.add(observer);
    }

    public void setMessage(String message) {
        this.message = message;
        notifyAllObservers();
    }

    private void notifyAllObservers() {
        for (MessageObserver observer : observers) {
            observer.updateMessage(message);
        }
    }
}
```

CHAPTER

THREE

UNIT TESTING

3.1 Software testing

3.1.1 What is software testing?

According to the ANSI/IEEE standard 610.12-1990, *testing* is *“the process of operating a system or component under specified conditions, observing or recording the results and making an evaluation of some aspect of the system or component.”* More informally, testing means verifying that software (or hardware) does what we expect it to do.

As an example, let’s assume we have written a method to calculate the quotient `a/b` of two natural numbers `a` and `b`:

```
static int division(int a, int b) { ... }
```

Now, we want to know whether our implementation is correct. We can call the method with arguments 6 and 3 and maybe we get 2 as result. This seems to be correct. Then we call the method with arguments 12 and 3, and we get 4. This looks fine, too. Finally, we call the method with arguments 5 and 2 and we get 3. Is that correct? Or did we expect the result to be 2? And what should be the result if the arguments are 4 and 0?

As the above example shows, tests are only useful if we have defined what our program is supposed to do. There are different ways to specify the expected behavior of software:

1. We could write a formal specification. In our example, this is relatively easy because the method performs a simple operation. Our formal specification could be:

   $$division(a, b) = \begin{cases} \lceil \frac{a}{b} \rceil & ifb \neq 0 \\ error\ exception, & otherwise \end{cases}$$

   According to this specification (note the $\lceil \cdot \rceil$), it becomes clear that the method should return 3 when called with arguments 5 and 2, and it should throw an exception if the second argument is 0.

2. Especially for more complex programs that are difficult to describe in a formal way, the specification is often written in the form of a text document with sentences like *“The method returns the quotient of two natural numbers…”*.

3. Finally, when working with customers, we often have to start with a list of *user requirements*, i.e., a description of what is need (*“The program should calculate a/b”*), and we have to write our own specification from that.

Requirements can come in all kinds of forms. In general, we can distinguish two broad categories of requirements:

- **Functional requirements** describe **what** the final product must be able to do. Examples:
  - “The program should calculate *a/b*”
  - “The program should sort a list”

  - "The program should print the first five prime numbers"
- **Non-functional requirements** describe **how** the product should perform. Examples:
  - "The program should be easy to use"
  - "The program should not need more than 10 milliseconds to calculate *a/b*"
  - "The program should be secure"

3.1.2 What can be tested?

Tests can be performed at different levels. Our above test of the `division()` method is a *unit test* because methods are the smallest entities in Java that we can independently test. We could also test an entire class or package (this is called a *module test*), several packages (*integration test*), or an entire program (*system test*). In this book, we will only do unit tests.

In general, the larger and the more complex the program you are testing, the more complicated the tests will be, and, more importantly, the more time-consuming it will be to fix a bug. A unit test can be done as soon as we write the method, and if we find a bug we can change the implementation of the method relatively easily. Imagine if we first finished writing the whole program and then, after several months of development, found during a system test that the program violates the specification! In the worst case, this could mean rewriting the whole program.

For this reason, it makes sense to start testing as early as possible in the form of unit tests. However, it should be noted that unit tests do not replace the other tests, because some errors appear only when we combine several methods or classes.

3.1.3 How to plan a unit test

We test software because we want to verify that it produces the correct results (functional requirements) in the right way (non-functional requirements). In general, software like our above `division()` method takes one or more input values and produces one or more result values. Unfortunately, in many cases there are many possible input values and testing our software with all of them would take a lot of time.

In practice, we can only test our software on a few input values and hope that the software also works correctly for other input values. For example, when we call our `division()` method with the arguments 6 and 3 and we get the correct result, we assume that the method also works correctly with the arguments 12 and 6. On the other hand, if the result for 6 and 3 is not correct, we know that our software contains a bug that we have to fix.

Does that mean that we should test our `division()` method with some random numbers? No, we can do something smarter than that:

1. First, we determine the *input domain* of the method, i.e., the set of possible input values. Because the method has two `int` parameters, the input domain is $\{-2^{31}..2^{31}-1\} \times \{-2^{31}..2^{31}-1\}$.
2. Then we split the input domain into interesting sub-domains. For example, we could decide it is interesting to test whether our method works correctly for $b = 0$. In that case we would obtain two sub-domains:
   - With $b = 0$: $\{-2^{31}..2^{31}-1\} \times \{0\}$
   - With $b \neq 0$: $\{-2^{31}..2^{31}-1\} \times \{-2^{31}..2^{31}-1\} \setminus \{0\}$
3. Instead of testing all possible input values, we choose a few input values from each subdomain for the tests, for example $a = 3, b = 0$ for the first sub-domain, and $a = 6, b = 3$ for the second sub-domain.

This approach also works for more complex methods. Let's take the following one:

```java
// returns a sorted array with the elements
// of "a" in ascending order
static int[] sortArray(int[] a)
```

Clearly, the input domain is very large. It contains all arrays of any length $n \geq 0$ containing all possible integer values. Possible sub-domains could be:

1. The empty array ($n = 0$)
2. Arrays with one element ($n = 1$)
3. Arrays containing random unsorted numbers for $n > 1$
4. Arrays containing numbers that have been already sorted, like `{1,2,3,4}`, for $n > 1$
5. Arrays containing numbers that have been already sorted in descending order, like `{4,3,2,1}`, for $n > 1$

It's always good to have disjoint sub-domains that cover the entire input domain. Our second sub-domain already covers the case $n = 1$, therefore it is not necessary to cover that case again in sub-domains 3 to 5.

3.2 Test coverage

3.2.1 Black box vs white box testing

Because the input domain of any non-trivial program is so large, identifying interesting input values for testing is a major challenge. If we do not have access to the source code of the program to test, we can only select the test values based on our experience and the specification. Such a test is called a *black box test* because the program that we want to test is like a opaque black box.

But if we have the source code of the program available, and that is the assumption in this book, we can use it to choose reasonable test values. This is a *white box test*.

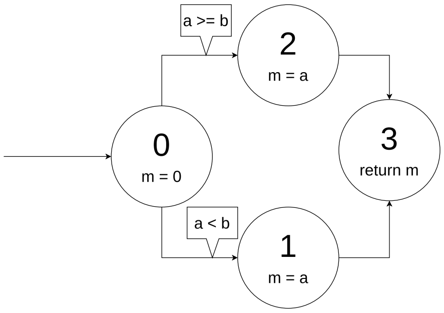
3.2.2 Control Flow Graph and node coverage

The following example shows an implementation of the `min()` method that contains a bug:

```java
// returns the minimum of "a" and "b"
static int min(int a, int b) {
    int m;
    if (a < b)
        m = a;
    else
        m = a;    // oops. That should be "m = b"
    return m;
}
```

If we call the above method with two numbers a and b where $a<b$ (for example, $a=3$ and $b=5$), we get always the correct result because the statement that contains the bug is never executed. The obvious truth is that **we can only find a bug in a program if the program reaches the faulty location in the code with our test values**. The conclusion here is that our input values should be chosen such that both branches of the if-else statement are tested.

We can visualize this by the *Control Flow Graph* (CFG) of the above code:



In the above control flow graph the node 0 represents the beginning of the method, the node 1 and 2 represent the two assignments in the if-else statement, and node 3 represents the `return` statement of the method.

In the Control Flow Graph (CFG), the small circles (called "nodes", French "nœuds") represent the beginning of the method and the statements. The arrows between the circles (called "edges", French "arêtes") represent how the program can go from statement to statement.

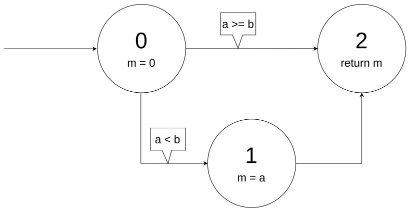
If we test the code with test values *a=3* and *b=5*, the program will go through the nodes 0, 1, and 3 of the CFG. To find the bug, we have to use test values where the program goes through node 2, for example *a=5* and *b=3*. With these two tests, we have covered all nodes of the CFG. We call this *100% node coverage*.

3.2.3 Edge coverage

While 100% node coverage is an important goal in testing, it does not necessarily mean that a program contains no bugs. Consider the following faulty implementation of the the `min()` method:

```java
static int min(int a, int b) {
    int m = 0;
    if (a < b) {
        m = a;
    }               // oops, we forgot the "else"
    return m;
}
```

Here is the CFG of the method:



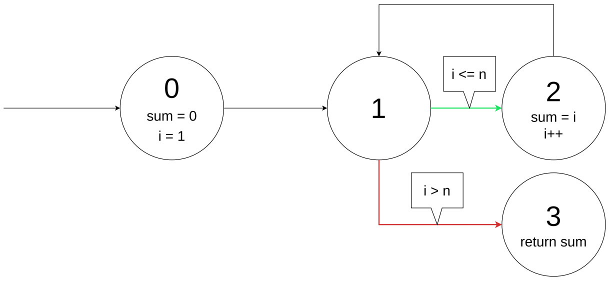
If we test this method with the test values *a=3* and *b=5*, the program will go through the nodes 0, 1, and 2, and we have achieved 100% node coverage without finding the bug! The bug only becomes visible if we use test values that force the program to go directly from node 0 to 2.

The above example shows that covering 100% of the *nodes* of the CFG with our tests is not enough. We have to choose our test values such that all all *edges* of the CFG are covered, too.

3.2.4 Path coverage

Unfortunately, 100% edge coverage is still not enough to find all bugs. The following example shows a method with a loop:

```java
// returns the sum of the values 1 to n
static int sum(int n) {
    int sum = 0;
    int i = 1;
    while (i <= n){
        sum = i;       // oops, this should be "sum += i"
        i++;
    }
    return sum;
}
```



A test with *n=0* covers the edges 0 → 1 and 1 → 3 and we get the correct result 0. A test with *n=1* covers the edges 0 → 1, 1 → 2, 2 → 1, and 1 → 3 and we get the correct result 1. With these two tests, we have covered all edges, but we have not found the bug.

To be sure that our program is correct, we would have to test all possible *paths* through the code:

- For *n=0*, the program takes the path 0 → 1 → 3 through the code.
- For *n=1*, the program takes the path 0 → 1 → 2 → 1 → 3 through the code.
- For *n=2*, the program takes the path 0 → 1 → 2 → 1 → 2 → 1 → 3 through the code. In this path, the bug becomes visible.
- etc.

In practice, 100% path coverage is not feasible if a program contains loops or recursion because there are too many possible paths. In practice, we are often satisfied with 100% node coverage or 100% edge coverage.

3.2.5 “Hidden” paths

Be aware that code can sometimes contain execution paths that are not directly visible in the source code. For example, the following statement looks like a simple assignment:

```
int r = a / b;
```

However, we know that Java programs can throw exceptions, and in this case, the division will throw an exception if *b=0*. Therefore, the code can be better understood as:

```
if (b == 0)
    throw new ArithmeticException();
else
    r = a / b;
```

Thus, if your goal is a test with 100% coverage, you also have to consider the test case *b=0*.

3.2.6 Coverage test tools

JaCoCo is a tool (and library) to perform coverage tests for Java programs: https://www.jacoco.org/jacoco/. When you run a program with JaCoCo, it calculates two metrics:

- JVM bytecode instruction coverage: this is similar to node coverage, but JaCoCo counts JVM bytecode instructions, not Java statements. A statement like a=b+2 corresponds to 4 JVM bytecode instructions.
- Branch coverage: this is similar to edge coverage, but only for the edges of if-else and switch statements.

Similar tools also exist for other programming languages. They help to check whether you have enough test cases.

3.3 Automated Unit Testing

3.3.1 Writing tests as a program

Testing is a repetitive task. In unit testing, we have to test every new method we write. And we have to repeat the test every time we changed the code of a method. It is therefore an obvious question whether we cannot let the computer do the testing.

As an example, consider again the min() method:

```
class Main {
    static int min(int a, int b) {
        ...
    }
}
```

We can write a test program to call this method and verify that the result is correct. The combination of test input values and the expected result is called a *test case*. In the following test code, we have two test cases:

```
// test case 1
int result1 = min(3,5);
if (result1 != 3) {
    System.out.println("Test 1 failed: Minimum of 3 and 5 should be 3");
}
```

(continues on next page)

(continued from previous page)

```java

// test case 2
int result2 = min(5,3);
if (result2 != 3) {
    System.out.println("Test 2 failed: Minimum of 5 and 3 should be 3");
}
```

The advantage of having a test program is that we can run the test automatically every time we change something in our project. There are even people who say that it is better to write *first* the tests and then the actual program! This practice is called *Test Driven Development* (TDD).

3.3.2 JUnit

Fortunately there are already tools and libraries to write tests. For Java, the most famous one is JUnit. Similar tools also exist for other programming language.

JUnit provides many useful classes and methods to write tests. To write a test you create a new class (for example, `MainTest`) and write a method for each test case. Depending on which version of JUnit you use, your test code will look different. In JUnit version 4, our above two tests of the `min()` method can be written like this:

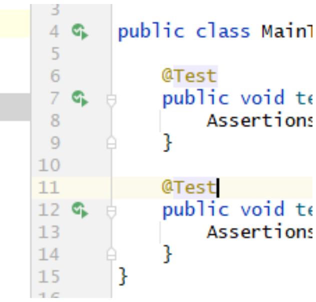
```java
import static org.junit.Assert.*;

public class MainTest {
    @org.junit.Test
    public void testFirstNumberLessThanSecondNumber() {
        assertEquals("Minimum of 3 and 5 should be 3", 3, Main.min(3,5));
    }

    @org.junit.Test
    public void testFirstNumberGreaterThanSecondNumber() {
        assertEquals("Minimum of 5 and 3 should be 3", 3, Main.min(5,3));
    }
}
```

In JUnit version 5, the two tests are written slightly differently:

```java
import org.junit.jupiter.api.Assertions;
import org.junit.jupiter.api.Test;

public class MainTest {
    @Test
    public void testFirstNumberLessThanSecondNumber() {
        Assertions.assertEquals(3, Main.min(3, 5), "Minimum of 3 and 5 should be 3");
    }

    @Test
    public void testFirstNumberGreaterThanSecondNumber() {
        Assertions.assertEquals(3, Main.min(5, 3), "Minimum of 5 and 3 should be 3");
    }
}
```

The method `assertEquals()` of the class `Assertions` takes three arguments: the expected value, the actual value produced by your implementation, and an (optional) message that is shown if the test fails, i.e., if the actual value and the expected value are not equal.

The `@Test` written above the two test methods is called an *annotation* and helps JUnit to find the methods that it should call to perform the tests. IntelliJ also uses them to show you the small green triangles that you can click to run individual tests (or all tests):



```
public class MainT

    @Test
    public void te
        Assertions
    }

    @Test
    public void te
        Assertions
    }
}
```

The class `Assertions` has many other methods to compare results, such as `assertArrayEquals()` for arrays, and `assertNotEquals()` to test for inequality. It is important to note that these methods use the `equals()` method when comparing objects. If you want to compare references, you have to use `assertSame()`. Check the documentation at https://junit.org/junit5/docs/5.0.1/api/org/junit/jupiter/api/Assertions.html.

3.3.3 Practical aspects of unit testing

The main idea behind unit testing is that your program is organized in small units that can be individually tested. As already said, in Java, methods can be seen as such units. However, if a method is very complex or does many different things, it becomes more difficult to test. As an example, consider the following (incomplete) code:

```
class DifficultToTest {
    static int m(int v1) {
        ...something complex using v1 to calculate v2...
        int v2 = ...
        ...something complex using v2 to calculate the result...
        int result = ...
        return result;
    }
}
```

As a developer, we would like to know whether the intermediate value `v2` and the result are correctly calculated. To do this with a unit test, it would be better to split the method in two:

```
class EasierToTest {
    static int m1(int v1) {
        ...something using v1 to calculate v2...
        int v2 = ...
        return v2;
    }

    static int m2(int v2) {
        ...something using v2 to calculate the result...
        int result = ...
        return result;
    }

    static int m(int v1) {
```

(continues on next page)

(continued from previous page)

```java
        int v2 = m1(v1);
        int result = m2(v2);
        return result;
    }
}
```

This new code is not only easier to read but also easier to test because you can provide your own values `v1` and `v2` to test the two parts of the calculation independently.

Another practical problem is the testing of non-static methods or methods that need objects as parameters. Consider the following class:

```java
class Employee {
    private int salary;

    public Employee(int s) { salary = s; }
    public void increaseSalary(int s) { salary += s; }
    public int getSalary() { return salary; }
}
```

When testing non-static methods like `increaseSalary()`, your test needs to “prepare” an object before the method can be called. In JUnit v5, the test code could look like this:

```java
public class EmployeeTest {
    @Test
    void testSalaryIncrease() {
        Employee employee = new Employee(1000);
        employee.increaseSalary(500);
        Assertions.assertEquals(1500, employee.getSalary());
    }
}
```

Although this test is correctly implemented, it’s difficult to see where the bug is located if the test fails. Did `increaseSalary()` not work correctly? Or was the bug in the constructor or in the `getSalary()` method?

There are different ways to address this problem. One is to add more test cases, for example for the construction of the object:

```java
public class EmployeeTest {
    @Test
    void testConstruction() {
        Employee employee = new Employee(1000);
        Assertions.assertEquals(1000, employee.getSalary());
    }

    @Test
    void testSalaryIncrease() {
        Employee employee = new Employee(1000);
        employee.increaseSalary(500);
        Assertions.assertEquals(1500, employee.getSalary());
    }
}
```

Alternative, we could do more tests inside one test case:

```java
public class EmployeeTest {
    @Test
    void testSalaryIncrease() {
        Employee employee = new Employee(1000);
        Assertions.assertEquals(1000, employee.getSalary());

        employee.increaseSalary(500);
        Assertions.assertEquals(1500, employee.getSalary());
    }
}
```

One can argue about what is the “right” way. Some developers prefer simple test methods, in which exactly one thing is tested. Others don’t like too many small trivial tests. We don’t want to get involved in this discussion and leave it to you to decide.

CHAPTER

FOUR

ALGORITHMS AND DATA STRUCTURES

4.1 Time Complexity

In the rapidly evolving world of computer science, the efficiency of an algorithm is paramount. As we strive to tackle increasingly complex problems and manage growing volumes of data, understanding how our algorithms perform becomes more important than ever.

This is where the concept of *time complexity* comes into play.

Time complexity provides a theoretical estimation of the time an algorithm requires to run relative to the size of the input data. In other words, it allows us to predict the efficiency of our code before we even run it. It's like having a magic crystal ball that tells us how our algorithm will behave in the wild!

Let's delve into the intricacies of time complexity and uncover the beauty and elegance of efficient code by studying first a very simple `sum()` method that calculates the total sum of all the elements in an integer array provided as argument.

```java
public class Main {
    public static int sum(int [] values) {
        int total = 0;
        for (int i = 0; i < values.length; i++) {
            total += values[i];
        }
        return total;
    }
}
```

One can measure the time it takes using `System.currentTimeMillis()` method that returns the current time in milliseconds since the Unix Epoch (January 1, 1970 00:00:00 UTC). It is typically used to get a timestamp representing the current point in time. Here is an example of how to use it to measure the time of one call to the `sum()` method.

```java
public class Main {
    public static void main(String[] args) {
        int[] values = {1, 2, 3, 4, 5};
        long startTime = System.currentTimeMillis();
        int totalSum = sum(values);
        long endTime = System.currentTimeMillis()
        long duration = (endTime - startTime);  // duration in milliseconds
    }
}
```

Now, if one makes vary the size of values one can observe the evolution of execution time in function of the size of the input array given in argument to `sum()` and plot it. Here is what we obtain on a standard laptop.

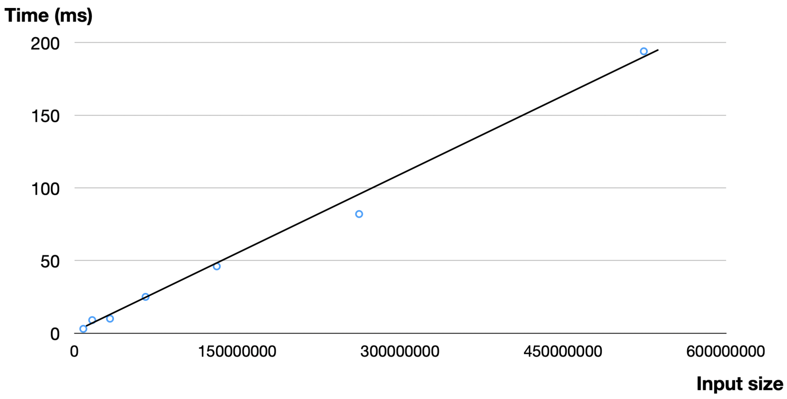


Fig. 1: Evolution of time measures taken by *sum* on arrays of increasing size.

Undoubtedly, the absolute time is heavily reliant on the specifications of the machine the code is executed on. The same code running on a different laptop could produce different timing results. However, it is noteworthy that the time evolution appears to be linear with respect to the array size, as illustrated by the trend line.

A crucial question arises: could this have been foreseen without even running the code? The answer is affirmative! Hartmanis and Stearns [HS65] laid down the foundations for such theoretical analyses from the source-code without (or pseudo-code, as the algorithm itself is of greater significance), even without requiring running the code and measure time. This great invention is explained next, but first things first, we need a simple computation model.

4.1.1 The Random Access Machine (RAM) model of computation

The RAM, or Random Access Machine, model of computation is a theoretical model of a computer that provides a mathematical abstraction for algorithm analysis. In the RAM model, each "simple" operation (such as addition, subtraction, multiplication, division, comparison, bitwise operations, following a reference, or direct addressing of memory) can be done in a single unit of time. It assumes that memory accesses (like accessing an element in an array: `value[i]` above) take constant time, regardless of the memory location. This is where the name "random access" comes from, since any memory location can be accessed in the same amount of time.

This abstraction is quite realistic for many practical purposes, and closely models real computers (a bit like Newton laws is a good approximation of general relativity).

Of course we can't assume a loop is a "simple" operation in the RAM model. One need to count the number of times its body will be executed. The next code add comments on the number of steps required to execute the sum algorithm.

```java
public static int sum(int [] values) {          // n = values.length
    int total = 0;                              // 1 step
    for (int i = 0; i < values.length; i++) {
        total += values[i];                     // 2* n steps (one memory access and one addition executed n times)
    }
    return total;                               // 1 step
}                                               // TOTAL: 2n + 2 steps
```

In practice, it is difficult to translate one step into a concrete time since it depends on many factors (machine, language, compiler, etc.). It is also not true that every operation takes exactly the same amount of time. Remember that it is just an approximation. We'll further simplify our step-counting approach by utilizing classes of functions that easily interpretable for practitioners like us.

Let us first realize in the next section that even for a consistent input size, the execution time of an algorithm can vary significantly.

4.1.2 Best-case and worst-case scenarios of an algorithm

Different inputs of the same size may cause an algorithm to take more or fewer steps to arrive at a result.

To illustrate this, consider the `linearSearch()` method that looks whether an array contains a specific target value and that returns the first index having this value, or `-1` if this value is not present in the array.

```java
/**
 * This method performs a linear search on an array.
 *
 * @param arr The input array.
 * @param x   The target value to search for in the array.
 * @return The index of the target value in the array if found,
 *         or -1 if the target value is not in the array.
 */
public static int linearSearch(int[] arr, int x) {
    for (int i = 0; i < arr.length; i++) {
        if (arr[i] == x) {
            return i;
        }
    }
    return -1;
}
```

This method achieves its goal by iterating through the array and returning the index of the first occurrence of the target value. If the target value isn't present, it returns `-1`.

In this case, the number of steps the `linearSearch()` method takes to complete is heavily dependent on the position of the target value within the array. If the target value is near the beginning of the array, the `linearSearch()` method completes quickly. We call this the *best-case scenario*.

Conversely, if the target value is at the end of the array or not present at all, the method must iterate through the entire array, which naturally takes more steps. We call this, the *worst-case scenario*.

The execution of `linearSearch()` can thus greatly vary depending not only on the *size* of the input array, but also on the *content* of the input array. Other categories of algorithms will have a execution that is mostly determined by the input size, rather than the input content. This characteristic is exemplified by the `sum()` method we previously analyzed.

The notation we are about to introduce for characterizing the execution time of an algorithm will allow us to represent both the best and worst-case scenarios.

4.1.3 The Big-O, Big-Omega and Big-Theta classes of functions

Let us assume that the number of steps an algorithm requires can be represented by the function $T(n)$ where n refers to the size of the input, such as the number of elements in an array. While this function might encapsulate intricate details about the algorithm's execution, calculating it with high precision can be a substantial undertaking, and often, not worth the effort.

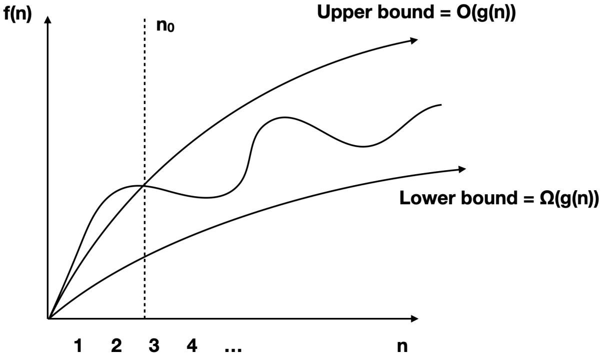
For sufficiently large inputs, the influence of multiplicative constants and lower-order terms within the exact runtime is overshadowed by the impact of the input size itself. This leads us to the concept of asymptotic efficiency, which is particularly concerned with how an algorithm's running time escalates with an increase in input size, especially as the size of the input grows unboundedly.

Typically, an algorithm that is asymptotically more efficient will be the superior choice for all but the smallest of inputs. This section introduces standard methods and notations used to simplify the asymptotic analysis of algorithms, thereby making this complex task more manageable. We shall see asymptotic notations that are well suited to characterizing running times no matter what the input.

Those so-called Big-Oh notations are sets or classes of functions. We have classes of function asymptotically bounded by above, below or both:

- $f(n) \in \mathcal{O}(g(n)) \iff \exists c \in \mathbb{R}^+, n_0 \in \mathbb{N} : f(n) \leq c \cdot g(n) \; \forall n \geq n_0$ (upper bound)
- $f(n) \in \Omega(g(n)) \iff \exists c \in \mathbb{R}^+, n_0 \in \mathbb{N} : f(n) \geq c \cdot g(n) \; \forall n \geq n_0$ (lower bound)
- $f(n) \in \Theta(g(n)) \iff \exists c_1, c_2 \in \mathbb{R}^+, n_0 \in \mathbb{N} : c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n) \; \forall n \geq n_0$ (exact bound)

What is common in the definitions of these classes of function is that we are not concerned about small constant. Instead we care about the big-picture that is when n becomes really large (say 10,000 or 1,000,000). The intuition for those classes of function notations are illustrated next.



One big advantage of Big-Oh notations is the capacity to simplify notations by only keeping the fastest growing term and taking out the numerical coefficients. Let us consider an example of simplification: $f(n) = c \cdot n^a + d \cdot n^b \quad$ with $a \geq b \geq 0$ and $c, d \geq 0$. Then we have $f(n) \in \Theta(n^a)$. This is even true if c is very small and d very big!

The simplification principle that we have applied are the following: $\mathcal{O}(c \cdot f(n)) = \mathcal{O}(f(n))$ (for $c > 0$) and $\mathcal{O}(f(n) + g(n)) \subseteq \mathcal{O}(\max(f(n), g(n))))$. You can also use these inclusion relations to simplify: $\mathcal{O}(1) \subseteq \mathcal{O}(\log n) \subseteq \mathcal{O}(n) \subseteq \mathcal{O}(n^2) \subseteq \mathcal{O}(n^3) \subseteq \mathcal{O}(c^n) \subseteq \mathcal{O}(n!)$

As a general rule of thumb, when speaking about the time complexity of an algorithm using Big-Oh notations, you must simplify if possible to get rid of numerical coefficients.

..Recursive Algorithms .. ——————-

4.1.4 Practical examples of different algorithms

To grasp a theoretical concept such as time complexity and Big O notation, concrete examples are invaluable. For each of the common complexities, we present an algorithmic example and then break down the reasons behind its specific time complexity. The following table provides an overview of the most prevalent complexity classes, accompanied by algorithm examples we explain after.

Complexity (name)	Algorithm
$\mathcal{O}(1)$ (constant)	Sum of two integers
$\mathcal{O}(\log n)$ (logarithmic)	Find an entry in a sorted array (binary search)
$\mathcal{O}(n)$ (linear)	Sum elements or find an entry in a not sorted array
$\mathcal{O}(n \log n)$ (linearithmic)	Sorting efficiently an array (merge sort)
$\mathcal{O}(n^2)$ (quadratic)	Sorting inefficiently an array (insertion sort)
$\mathcal{O}(n^3)$ (cubic)	Enumerating all possible triples taken from an array
$\mathcal{O}(2^n)$ (exponential)	Finding elements in an array summing to zero (Subset-sum)
$\mathcal{O}(n!)$ (factorial)	Visiting all cities in a country by minimizing the distance

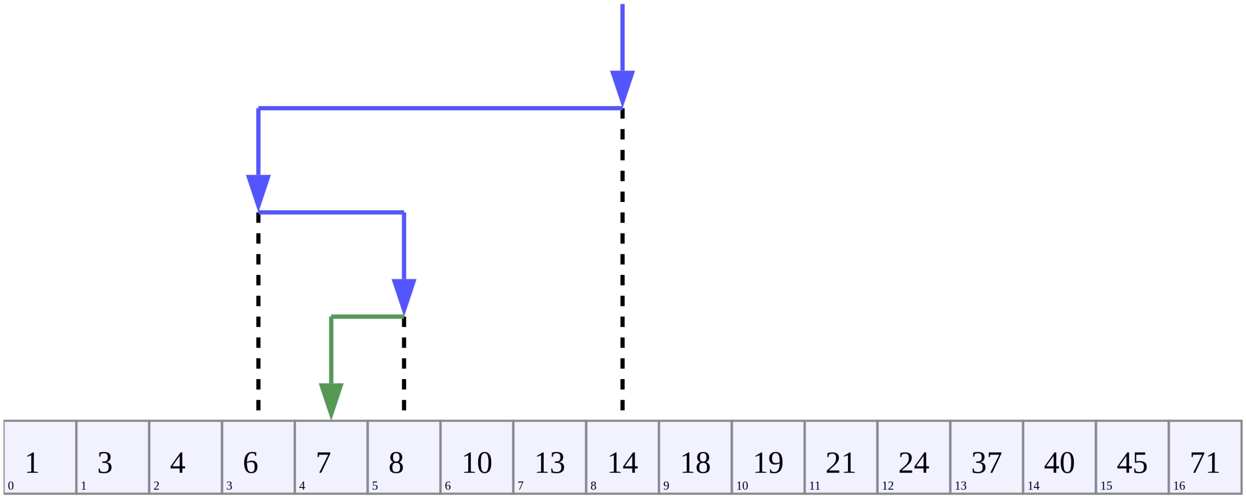
Binary Search

The Binary search, also known as dichotomic search, is a search algorithm that finds the position of a target value within a sorted array. It works by halving the number of elements to be searched each time, which makes it incredibly efficient even for large arrays.

Here's how the binary search algorithm works:

1. You start with the middle element of the sorted array.
2. If the target value is equal to this middle element, then you've found the target and the algorithm ends.
3. If the target value is less than the middle element, then you repeat the search with the left half of the array.
4. If the target value is greater than the middle element, then you repeat the search with the right half of the array.
5. You keep repeating this process until you either find the target value or exhaust all elements.

The execution of this search is illustrated on next schema searching for value 7 repeating 4 times the process until finding it. On this array of 16 entries, the search will never require more than four trials, so this is the worst-case scenario.



This algorithm has a time complexity of $\mathcal{O}(\log n)$ because each time we go through the loop, the number of elements to be searched is halved and in the worst case, this process is repeated $\log n$ times. On the other hand, if we are lucky, the search immediately find the element at the first iteration. Therefore the best-case time complexity is $\Omega(1)$.

The Java code is a direct translation of the explanation of the algorithm.

```
    /**
     * This method performs a binary search on a sorted array.
     * The array remains unchanged during the execution of the function.
     *
     * @param arr The input array, which must be sorted in ascending order.
     * @param x   The target value to search for in the array.
     * @return The index of the target value in the array if found,
 *          or -1 if the target value is not in the array.
     */
public static int binarySearch(int arr[], int x) {
    int left = 0, right = arr.length - 1;
    while (left <= right) {
        int mid = left + (right - left) / 2;

        // Check if x is present at mid
        if (arr[mid] == x)
            return mid;

        // If x greater, ignore left half
        if (arr[mid] < x)
            left = mid + 1;

        // If x is smaller, ignore right half
        else
            right = mid - 1;
    }

    // If we reach here, then element was not present
    return -1;
}
```

Tip: Notice that the expression `left + (right - left) / 2` is preferred over the somewhat simpler `(left + right) / 2` to calculate the middle index in a binary search. At first glance, they seem to do the same thing, and indeed, they usually do give the same result. The main advantage of using `left + (right - left) / 2` over `(left + right) / 2` comes into play when you are dealing with large numbers. The problem with `(left + right) / 2` is that the sum of `left` and `right` could exceed the maximum limit of the integer in the Java language that is is $2^{31}-1$, causing an integer overflow, which can lead to unexpected results or errors. The one used `left + (right - left) / 2` does not have this overflow risk problem.

Tip: Keep in mind that when dealing with objects (as opposed to primitive types), we would want to use the `equals()` method instead of ==. This is because `equals()` tests for logical equality, meaning it checks whether two objects are logically equivalent (even if they are different instances). On the other hand, == tests for reference equality, which checks whether two references point to the exact same object instance. For objects where logical equality is more meaningful than reference equality, like `String` or custom objects, using `equals()` is the appropriate choice.

Linear Search

We already have seen the sum algorithm and its $\Theta(n)$ time complexity. Another example of a linear time complexity algorithm is the linear_search. The time complexity of the linear search algorithm is $\mathcal{O}(n)$, where *n* is the size of the array, because in the worst-case scenario (the target value is not in the array or is the last element in the array), the algorithm has to examine every element in the array once. In the best-case scenario for the linear search algorithm, the target value is the very first element of the array. Therefore, in the best-case scenario, the time complexity of the linear search algorithm is $\mathcal{O}(1)$ or we can simply say that the algorithm is also in $\Omega(1)$.

Merge Sort

Merge sort is a *divide-and-conquer* algorithm for sorting lists or arrays of items using pair-wise comparisons. It works by dividing the unsorted list into n sublists, each containing one element (a list of one element is considered sorted), and then repeatedly merging sublists to produce newly sorted sublists until there is only one sublist remaining.

Here's the basic idea behind merge sort:

- Divide: If the list is of length 0 or 1, then it is already sorted. Otherwise, divide the unsorted list into two sublists of about half the size.
- Conquer: Sort each sublist recursively by re-applying the merge sort.
- Combine: Merge the two sublists back into one sorted list.

Here is a simple implementation of Merge Sort in Java:

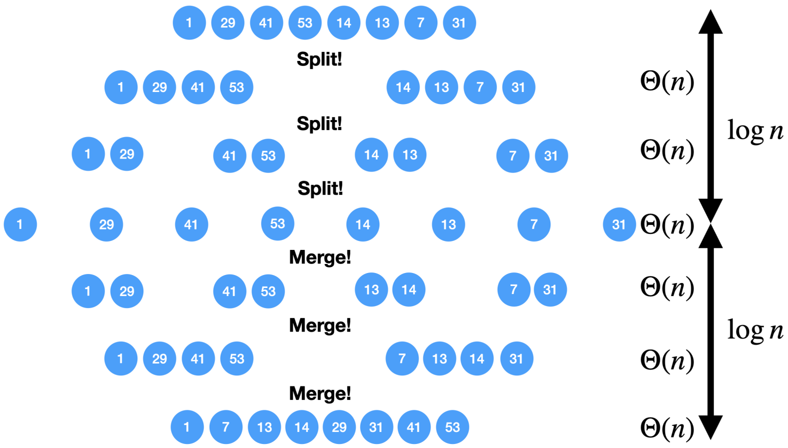
```java
private static void merge(int[] left, int [] right, int result[]) {
    assert(result.length == left.length + right.length);
    int index = 0, leftIndex = 0 , rightIndex = 0;
    while (leftIndex != left.length || rightIndex != right.length) {
        if (rightIndex == right.length ||
            (leftIndex != left.length && left[leftIndex] < right[rightIndex])) {
            result[index] = left[leftIndex];
            leftIndex++;
        }
        else {
            result[index] = right[rightIndex];
            rightIndex++;
        }
        index++;
    }
}

/**
 * Sort the values increasingly
 */
public static void mergeSort(int[] values) {
    if(values.length == 1) // list of size 1, already sorted
        return;

    int mid = values.length/2;

    int[] left = new int[mid];
    int[] right = new int[values.length-mid];
```

(continues on next page)

(continued from previous page)

```
    // copy values[0..mid-1] to left
    System.arraycopy(values, 0, left, 0, mid);
    // copy values[mid..values.length-1] to right
    System.arraycopy(values, mid, right, 0, values.length-mid);

    // sort left and right
    mergeSort(left);
    mergeSort(right);

    // merge left and right back into values
    merge(left, right, values);
}
```

The Merge sort is a divide and conquer algorithm. It breaks the array into two subarrays, sort them, and then merges these sorted subarrays to produce a final sorted array. All the operations and the data-flow of execution is best understood with a small visual example.



There are $\Theta(\log n)$ layers of split and merge operations. Each layer requires $\Theta(n)$ operations by summing all the split/merge operations at one level. In the end, the time complexity of the merge sort algorithm is the product of the time complexities of these two operations that is $\Theta(n \log n)$.

Insertion Sort

The insertion sort algorithm is probably the one you use when sorting a hand of playing cards. You start with one card in your hand (the sorted portion). For each new card, you insert it in the correct position in your hand by moving over any cards that should come after it.

The Java code is given next.

```
/**
 * This method sort the array using Insertion Sort algorithm.
 *
 * @param arr The input array.
 */
public static void insertionSort(int[] arr) {
    for (int i = 1; i < arr.length; i++) {
        int key = arr[i];
```

(continues on next page)

(continued from previous page)

```
        int j = i - 1;
        // Move elements of arr[0..i-1], that are greater than key,
        // to one position ahead of their current position
        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j = j - 1;
        }
        arr[j + 1] = key;
    }
}
```

For each element (except the first), it finds the appropriate position among the already sorted elements (all elements before the current one), and inserts it there by moving larger elements up. Moving the larger elements up is the goal of the inner `while` loop.

The time complexity of insertion sort is $\mathcal{O}(n^2)$ in the worst-case scenario, because each of the `n` elements could potentially need to be compared with each of the `n` already sorted elements. However, in the best-case scenario (when the input array is already sorted), the time complexity is $\mathcal{O}(n)$, because each element only needs to be compared once with the already sorted elements. Alternatively, we can simply say that the insertion sort algorithm runs in $\Omega(n)$ and $\mathcal{O}(n^2)$.

Triple Sum

We consider a algorithm that checks if there exists at least one combination of three elements in an array that sum up to zero. Here an implementation in Java:

```
/**
 * This method checks if there are any three numbers in the array that sum up to zero.
 *
 * @param arr The input array.
 * @return True if such a triple exists, false otherwise.
 */
public static boolean checkTripleSum(int[] arr) {
    int n = arr.length;

    for (int i = 0; i < n - 2; i++) {
        for (int j = i + 1; j < n - 1; j++) {
            for (int k = j + 1; k < n; k++) {
                if (arr[i] + arr[j] + arr[k] == 0) {
                    return true;
                }
            }
        }
    }

    return false;
}
```

In this program, `checkTripleSum()` goes through each possible combination of three elements in the input array. If it finds a triple that sums up to zero, it immediately returns `true`. If no such triple is found after checking all combinations, it returns `false`. Since there are $n * (n - 1) * (n - 2)/6$ possible combinations of three elements in an array of length n, and we're checking each combination once, the time complexity of this method is $\mathcal{O}(n^3)$ and $\Omega(1)$. The best-case scenario occurs if the first three elements in the array sum to zero so that each loop is in its first iteration when the `return` instruction occurs.

Subset-Sum

The subset-sum problem is a classic problem in computer science: Given a set of integers, is there a subset of the integers that sums to zero? This is a generalization of the `checkTripleSum()` problem we have seen before, in which the allowed subsets must not contain exactly 3 elements.

The algorithm we will use for solving the problem is a *brute-force* approach that will enumerate all the possible subsets. A common approach to enumerate all the subsets is to use recursion. We can consider each number in the set and make a recursive call for two cases: one where we exclude the number in the subset, and one where we include it.

The Java code is given next. It calls an auxiliary method with an additional argument `sum()` that is the sum of the elements up to index `i` already included.

```java
/**
 * This method checks if there is a subset of the array that sums up to zero.
 *
 * @param arr   The input array.
 * @return True if there is such a subset, false otherwise.
 */
public static boolean isSubsetSumZero(int[] arr) {
    return isSubsetSum(arr, 0, 0) || ;
}

private static boolean isSubsetSum(int[] arr, int i, int sum) {
    // Base cases
    if (i == arr.length) { // did not find it
        return false;
    }
    if (sum + arr[i] == 0) { // found it
            return true;
    } else {
            // Check if sum can be obtained by excluding / including the next
            return isSubsetSum(arr, i + 1, sum) ||
                   isSubsetSum(arr, i + 1, sum + arr[i]);
    }
}
```

The time complexity of this algorithm is $\mathcal{O}(2^n)$, because in the worst case it generates all possible subsets of the array, and there are 2^n possible subsets for an array of n elements. The worst case is obtained when there is no solution and that `false` is returned. The best-case complexity is $\Omega(1)$ and is obtained when the first element in the array is zero so that the algorithm immediately returns `true`.

Note that this algorithm has an exponential time complexity (so far the algorithm we have studied were polynomial e.g., $\mathcal{O}(n^3)$). Therefore, although this approach will work fine for small arrays, it will be unbearably slow for larger ones.

Tip: The question that arises is: Can we find an efficient algorithm to solve this problem more efficiently? By “efficient”, we mean an algorithm that doesn’t take an exponential time to compute as the size of the input grows. The answer is, maybe but we don’t know yet. Researchers stumbled upon a category of problems discovered in the early 1970’s, that share a common trait: They all seem to be impossible to solve efficiently, but if you’re handed a potential solution, you can at least verify its correctness quickly. The subset-sum problem belongs to this class. This category is called *NP* (Nondeterministic Polynomial time).

Now, within NP, there's a special class of problems dubbed *NP-complete*. What is so special about them? Well, if you can find an efficient solution for one *NP-complete* problem, you've essentially found efficient solutions for all of them! The subset-sum problem is one of these NP-complete problems. Like its NP-complete siblings, we don't have efficient solutions for it yet. But remember, this doesn't mean that no efficient solution exists; we just haven't found one and it was also not yet proven that such an algorithm does not exist. This also doesn't mean that there are no faster algorithms for the subset-sum problem that the one we have shown. For instance a *dynamic programming* algorithm (out of scope of this introduction to algorithms) for subset-sum can avoid redundant work, but still has a worst-case exponential time complexity.

Visiting all cities in a country minimizing the distance

The Traveling Salesman Problem (TSP) is a classic NP-hard problem in the field of computer science and operations research. The problem is simple to state: Given a list of cities and the distances between them, find the shortest possible route that visits each city exactly once and returns to the starting city.

This problem, as well as its decision version (i.e., does there exist a circuit with a total length shorter than a given value?), is proven to be NP-complete. We suggest a straightforward brute-force approach to address this challenge. This method involves enumerating all possible permutations of the cities and maintaining a record of the permutation that yields the shortest distance. The time complexity of this strategy is $O(n!)$ (factorial) because it necessitates generating all the permutations and computing the total length for each one.

When the number of cities exceeds 12, the brute-force method becomes increasingly impractical. Even with high-speed modern computers, attempting to solve the TSP for, say, 20 cities using brute-force would involve evaluating $20! \sim 2.43 * 10^{18}$ routes—a task that would take many years.

Tip: In contrast, more sophisticated algorithms have been developed for the TSP. Techniques such as branch and bound can effectively solve TSP instances with thousands of cities, making them vastly more scalable than the brute-force approach.

```java
public class TSPBruteForce {

    public static void main(String[] args) {
        int[][] distanceMatrix = {
                {0, 10, 15, 20},
                {10, 0, 35, 25},
                {15, 35, 0, 30},
                {20, 25, 30, 0}
        };

        Result bestTour = findBestTour(distanceMatrix);

        System.out.println("Shortest Tour: " + bestTour.tour);
        System.out.println("Distance: " + bestTour.distance);
    }

    /**
     * Calculates the shortest tour that visits all cities.
     * @param distanceMatrix the distance matrix
     * @return the shortest tour
     */
    public static Result findBestTour(int [][] distanceMatrix) {
```

(continues on next page)

(continued from previous page)

```
        boolean[] visited = new boolean[distanceMatrix.length];
        // already fix 0 as the starting city
        visited[0] = true;
        List<Integer> currentTour = new ArrayList<>();
        currentTour.add(0);
        Result bestTour = findBestTour(visited, currentTour, 0, distanceMatrix);

        return bestTour;
    }

    private static Result findBestTour(boolean[] visited, List<Integer> currentTour, int␣
↪currentLength, int[][] distanceMatrix) {
        int lastCity = currentTour.get(currentTour.size() - 1);

        if (currentTour.size() == visited.length - 1) {
            currentLength += distanceMatrix[lastCity][0];  // return to city 0
            return new Result(new ArrayList<>(currentTour), currentLength);
        }

        Result bestResult = new Result(null, Integer.MAX_VALUE);

        for (int i = 1; i < visited.length; i++) {
            if (!visited[i]) {
                visited[i] = true;
                currentTour.add(i);
                int newLength = currentLength + distanceMatrix[lastCity][i];
                Result currentResult = findBestTour(visited, currentTour, newLength,␣
↪distanceMatrix);
                if (currentResult.distance < bestResult.distance) {
                    bestResult = currentResult;
                }
                currentTour.remove(currentTour.size() - 1);
                visited[i] = false;
            }
        }

        return bestResult;
    }



    static class Result {
        List<Integer> tour;
        int distance;

        Result(List<Integer> tour, int distance) {
            this.tour = tour;
            this.distance = distance;
        }
    }

}
```

Exercise

What is the time complexity of following algorithm? Characterize the best and worst case.

```
/**
 * Counts the minimum number of bits in the binary representation
 * of a positive input number. Example: 9 requires 4 bits (1001).
 * It halves it until it becomes zero counting the number of iterations.
 *
 * @param n The input number, which must be a positive integer.
 * @return The number of bits in the binary representation of the input number.
 */
public static int bitCount(int n) {
    int bitCount = 0;

    while (n > 0) {
        bitCount++;
        n = n >> 1;  // bitwise shift to the right, equivalent to dividing by 2
    }

    return bitCount;
}
```

4.2 Space complexity

Aside from the time, the memory is also a scarce resource that is worth analyzing for an algorithm. The *space complexity* of an algorithm quantifies the amount of space or memory taken by an algorithm to run, expressed as a function of the length of the input. Since this notion of "space" is subject to interpretation, let us separate it into two less ambiguous definitions.

- The *auxiliary space* is the extra space or the temporary space used by the algorithm during its execution.
- The *input space* is the space taken by the argument of the algorithm or the instance variables if any.

The definition of space complexity includes both: *space complexity* = *auxiliary space complexity* + *input space complexity*.

The next *sum* method computing the sum of the elements in an array uses $\mathcal{O}O(1)$ auxiliary space, but $\Theta(n)$ input space where n is the lenght of the input array. Its space complexity is thus $Theta(n)$.

```
public static int sum(int[] array) {
    int sum = 0;  // O(1) space for the sum variable
    for (int i = 0; i < array.length; i++) {
        sum += array[i];
    }
    return sum;
}
```

On the other hand, the method *range* uses $\Theta(n)$ auxiliary space where n is the absolute value of the input parameter, but it consumes only $\mathcal{O}(1)$ input space since an integer has a fixed size encoding of 32 bits in Java.

```java
public static int[] range(int n) {
    int[] result = new int[n];
    for(int i = 0; i < n; i++)
        result[i] = i;
    return result;
}}
```

Notice that we have chosen to express the space complexity in terms of the absolute value of the number. We could also have chosen to express it in terms of the input size. The input size is 32 bits here, but in binary form, it could encode a number as large as $2^{31} - 1$. Therefore, if k is the number of bits required to encode the value n , the auxiliary space complexity is $\mathcal{O}(2^k)$ (exponential thus). This illustrates the importance of clearly specifying what the parameters in a complexity expression represent when describing complexity.

4.2.1 Space complexity of recursive algorithms

Notice that the extra space may also take into account the space of the system stack in the case of a recursive algorithm. In such a situation, when the recursive call happens, the current local variables are pushed onto the system stack, where they wait for the call the return and unstack the local variables.

More exactly, If a method `A()` calls method `B()` (which can possibly be `A()` in case of recursion) inside it, then all the variables still in the scope of the method `A()` will get stored on the system stack temporarily, while the method `B()` is called and executed inside the method `A()`.

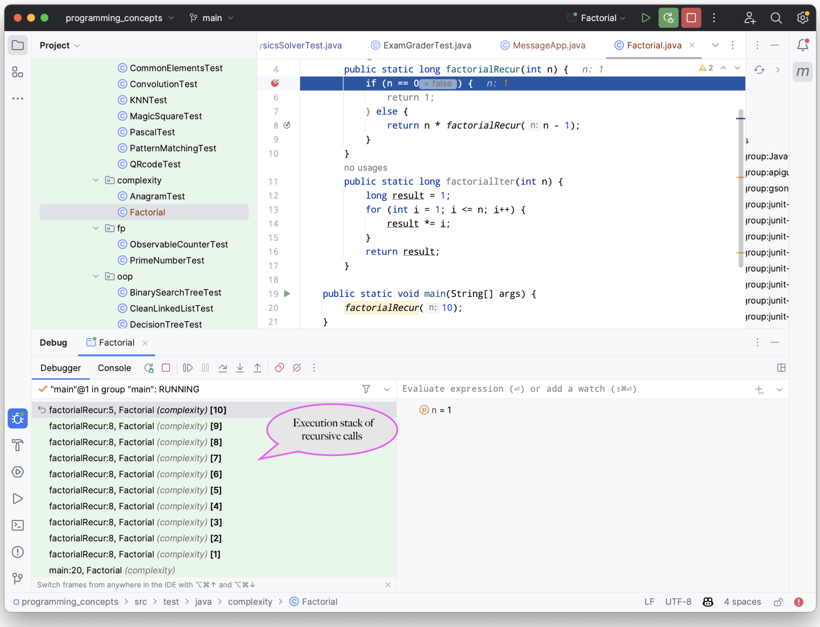
Let us compare the space and time complexity of an iterative and a recursive computation of the factorial of a number expressed in function of n, the value of the number for which we want to compute the factorial.

```java
public class Factorial {
    // Recursive implementation
    public static long factorialRecur(int n) {
        if (n == 0) {
            return 1;
        } else {
            return n * factorialRecur(n - 1);
        }
    }

    // Iterative implementation
    public static long factorialIter(int n) {
        long result = 1;
        for (int i = 1; i <= n; i++) {
            result *= i;
        }
        return result;
    }
}
```

Both implementations have a time complexity of $\Theta(n)$. However, the space complexity of the iterative version is $O(1)$, while the one of the recursive version is $\Theta(n)$.

You may be a bit surprised by this result since no array of size n is ever created in the recursive version. True! But a stack of size `n` is created. A stack? Yes, a stack, but it is not visible and it is created by the JVM.

Indeed, as explained before, every recursive call requires to store the local context or *frame* so that when the recursion returns, the multiplication can be performed. The sucessive frames are stored in a system stack that is transparently managed by the JVM, and that is part of the auxiliary space. The system stack for computing the factorial of 10 will look like `[10*[9*[8*[7*[6*[5*[4*[3*[2*[1]]]]]]]]]]`. This system stack can be visualized using the IntelliJ debugger by adding a breakpoint in the method. The call stack is shown at the bottom left of IntelliJ and you can see what the local context is by clicking on each *frame*.



```java
    public static long factorialRecur(int n) {  n: 1
        if (n == 0 = false ) {  n: 1
            return 1;
        } else {
            return n * factorialRecur( n: n - 1);
        }
    }
    no usages
    public static long factorialIter(int n) {
        long result = 1;
        for (int i = 1; i <= n; i++) {
            result *= i;
        }
        return result;
    }

public static void main(String[] args) {
    factorialRecur( n: 10);
}
```

```
"main"@1 in group "main": RUNNING
factorialRecur:5, Factorial (complexity) [10]
factorialRecur:8, Factorial (complexity) [9]
factorialRecur:8, Factorial (complexity) [8]
factorialRecur:8, Factorial (complexity) [7]
factorialRecur:8, Factorial (complexity) [6]
factorialRecur:8, Factorial (complexity) [5]
factorialRecur:8, Factorial (complexity) [4]
factorialRecur:8, Factorial (complexity) [3]
factorialRecur:8, Factorial (complexity) [2]
factorialRecur:8, Factorial (complexity) [1]
main:20, Factorial (complexity)
```

```
n = 1
```

Tip: It is quite frequent to have time complexity that is larger than the space complexity for an algorithm. But the opposite is not true, at least for the auxiliary space complexity. The time complexity is necessarily at least the one of the auxiliary space complexity, because you always need the same order as elementary steps as the one of the consumed memory.

Tip: When an uncaught exception occurs, you can also visualize the output, the execution stack of the successive calls from the main method up to the line of code that caused the exception to be thrown.

Improving the space complexity of merge sort

In the merge_sort_implem implementation, new arrays are created at each level of recursion. The overall space complexity is thus of $\mathcal{O}(n \log n)$, where n is the number of elements in the input array. This is because, at each level of the recursion, new arrays are created in the current frame, adding up to n elements per level, and the recursion goes $\log n$ levels deep.

The time complexity required by our merge sort algorithm can be lowered to $\mathcal{O}(n)$ for the auxiliary space. We can indeed create a single temporary array of size n once and reusing it in every merge operation. This temporary array requires n units of space, which is independent of the depth of the recursion. As such, the space complexity of this version of the merge sort algorithm is $\mathcal{O}(n)$, which is an improvement over the original version.

```java
public class MergeSort {

    private void merge(int[] arr, int[] temp, int leftStart, int mid, int rightEnd) {
```

(continues on next page)

(continued from previous page)

```java
        int leftEnd = mid;
        int rightStart = mid + 1;
        int size = rightEnd - leftStart + 1;

        int left = leftStart;
        int right = rightStart;
        int index = leftStart;

        while (left <= leftEnd && right <= rightEnd) {
            if (arr[left] <= arr[right]) {
                temp[index] = arr[left];
                left++;
            } else {
                temp[index] = arr[right];
                right++;
            }
            index++;
        }
    // copy rest of left side
    System.arraycopy(arr, left, temp, index, leftEnd - left + 1);
    // copy rest of right side
    System.arraycopy(arr, right, temp, index, rightEnd - right + 1);
    // copy temp back to original array
    System.arraycopy(temp, leftStart, arr, leftStart, size);
    }

    public void sort(int[] arr) {
        int[] temp = new int[arr.length];
        sort(arr, temp, 0, arr.length - 1);
    }

    private void sort(int[] arr, int[] temp, int leftStart, int rightEnd) {
        if (leftStart >= rightEnd) {
            return;
        }
        int mid = leftStart + (rightEnd - leftStart) / 2;
        sort(arr, temp, leftStart, mid);
        sort(arr, temp, mid + 1, rightEnd);
        merge(arr, temp, leftStart, mid, rightEnd);
    }

    public static void main(String[] args) {
        MergeSort mergeSort = new MergeSort();
        int[] arr = {38, 27, 43, 3, 9, 82, 10};
        mergeSort.sort(arr);
        for (int i : arr) {
            System.out.print(i + " ");
        }
    }
}
```

It is worth noting that in both versions of the algorithm, the time complexity remains the same: $\mathcal{O}(n \log n)$. This is because the time complexity of merge sort is determined by the number of elements being sorted (n) and the number of levels in the recursion tree ($\log n$), not by the amount of space used.

4.3 Algorithm correctness

A loop invariant is a condition or property that holds before and after each iteration of a loop. It is used as a technique for proving formally the correctness of an iterative algorithm. The loop invariant must be true:

1. Before the loop begins (Initialization).
2. Before each iteration (Maintenance).
3. After the loop terminates (Termination).

The loop invariant often helps to prove something important about the output of the algorithm.

The code fragment max_invariant_while illustrates a simple loop invariant for a method searching for the maximum of an array.

```java
/**
 * Finds the maximum value in the provided array.
 *
 * @param a The array of integers.
 * @return The maximum integer value in the array using while loop
 */
public static int max(int[] a) {
    int m = a[0];
    int i = 1;
    // inv: m is equal to the maximum value on a[0..0]
    while (i != a.length) {
        // inv: m is equal to the maximum value on a[0..i-1]

        if (m < a[i]) {
            m = a[i];
        }
        // m is equal to the maximum value on a[0..i]
        i++;
        // inv: m is equal to the maximum value on a[0..i-1]
    }
    // m is equal to the maximum value in the entire array a[0..a.length-1]
    return m;
}
```

The correctness of the `max()` algorithm is a consequence of the correctness of the invariant. When `for` loops are used instead of `while` loops, one generally only expresses the invariant before each iteration as shown next.

```java
public static int max(int[] a) {
    int m = a[0];
    int i = 1;
    // inv: m is equal to the maximum value on a[0..0]
    for (int i = 1; i < a.length; i++) {
        // inv: m is equal to the maximum value on a[0..i-1]
        if (m < a[i]) {
            m = a[i];
        }
```

(continues on next page)

(continued from previous page)

```java
        // m is equal to the maximum value on a[0..i]
    }
    // m is equal to the maximum value in the entire array a[0..a.length-1]
    return m;
}
```

In order to be complete, we also need to prove that invariant itself is correctly maintained:

- Initialization: When entering the loop, `i == 1`. The invariant is thus that `m` should contain the maximum of subarray with only the first element. Since the maximum of a single element is the element itself, the invariant holds when entering the loop.
- Maintenance: If `m` is the maximum value in `a[0..i-1]` at the start of the loop, the current maximum either remains `m` or it becomes `a[i]` during the iteration, ensuring it is the maximum of `a[0..i]` by the end of the iteration. So, the invariant holds for the next iteration as well.
- Termination: At the end of the loop, `i == a.length`, and based on our invariant, `m` holds the maximum value of `a[0..a.length-1]`, which means `m` is the maximum of the entire array, which proves the correctness of our algorithm.

Let us now rewrite the `max()` algorithm in a recursive form.

```java
/**
 * Finds the maximum value in the provided array.
 *
 * @param a The array of integers.
 * @return The maximum integer value in the array using while loop
 */
public static int max(int[] a) {
    return maxRecur(a, a.length-1);
}

/**
 * Finds the maximum value in the subarray.
 *
 * @param a The array of integers.
 * @param i The index, a value in [0..a.length-1].
 * @return The maximum value in the sub-array a[0..i]
 */
private static int maxRecur(int[] a, int i) {
    if (i == 0)
       return a[i];
    else
       return Math.max(maxRecur(a,i-1),a[i]);
}
```

The correctness of a recursive algorithm is done by induction. We do it on the inductive parameter `i`.

- Base case: proof that the algorithm is correct when the algorithm is not recursing (when `i == 0` here). When `i == 0` we have `max(a[0]) == a[0]`.
- Induction: Assuming the algorithm is correct for $i - 1$, we prove that the algorithm is correct for i. We have that `max(a[0],...,a[i-1],a[i]) == max(max(a[0],...,a[i-1]),max(a[i]))` (by associativity of `max` operation). Then we have `max(max(a[0],...,a[i-1]),max(a[i])) == max(max(a[0],..., a[i-1]),a[i])`. Assuming the first part is correct (this is our induction hypothesis), this expression is precisely the one we compute.

Exercise

Find an invariant for the loop of the bubble maxsum algorithm. Prove that the invariant is correctly maintained. Hint: Your invariant should express a property on the variables `maxCurrent` and `maxGlobal` with respect to index `i`. A good exercise is to write a recursive version of this algorithm and to write the specification of it.

```java
public class MaxSubArray {

    /**
     * Computes the sum of the maximum contiguous subarray.
     * Example:
     *   int[] nums = {-2, 1, -3, 4, -1, 2, 1, -5, 4};
     *   maxSubArray(nums);  // Returns 6, for subarray [4, -1, 2, 1].
     *
     * @param nums An array of integers.
     * @return The sum of the maximum subarray.
     *
     */
    public static int maxSubArray(int[] nums) {

        int maxCurrent = nums[0];
        int maxGlobal = nums[0];

        for (int i = 1; i < nums.length; i++) {
            // invariant
            maxCurrent = Math.max(nums[i], maxCurrent + nums[i]);
            maxGlobal = Math.max(maxGlobal, maxCurrent);
        }

        return maxGlobal;
    }

    public static void main(String[] args) {
        int[] nums = {-2, 1, -3, 4, -1, 2, 1, -5, 4};
        System.out.println(maxSubArray(nums));  // Outputs 6
    }
}
```

Exercise

Find an invariant for the outer loop of the bubble bubble_loop sort algorithm. Prove that the invariant is correctly maintained. Hint: Your invariant should express a property on the array with respect to index `i`. A subpart of the array is already sorted? What values are they?

```java
public class Main {
    public static void main(String[] args) {
        int[] numbers = {5, 1, 12, -5, 16};
        bubbleSort(numbers);

        for (int i = 0; i < numbers.length; i++) {
            System.out.print(numbers[i] + " ");
```

(continues on next page)

(continued from previous page)

```
        }
    }

    public static void bubbleSort(int[] array) {
        int n = array.length;
        for (int i = 0; i < n-1; i++) {
            // invariant
            for (int j = 0; j < n-i-1; j++) {
                if (array[j] > array[j+1]) {
                    // swap array[j] and array[j+1]
                    int temp = array[j];
                    array[j] = array[j+1];
                    array[j+1] = temp;
                }
            }
        }
    }
}
```

4.4 Abstract Data Types (ADT)

In the context of data collection, an Abstract Data Type (ADT) is a high-level description of a collection of data and of the operations that can be performed on this data.

An ADT can best be described by an interface in Java. This interface specifies what operations can be done on the data, but without prescribing how these operations will be implemented. Implementation details are abstracted away.

It means that the underlying implementation of an ADT can change without affecting how the users of the ADT interact with it.

Abstract Data Types are present in the Java Collections Framework. Let's consider the List interface that belongs to the standard `java.util` namespace. This is one of the most frequently used Abstract Data Types. It defines an ordered collection of elements, with duplicates allowed. `List` is an ADT because it specifies a set of operations (e.g., `add(E e)`, `get(int index)`, `remove(int index)`, `size()`) that you can perform on a list without specifying how these operations are concretely implemented.

To get a concrete implementation of a `List`, you must use one of the concrete classes that implement this interface, for instance ArrayList or LinkedList. Whatever the one you choose the high level contract described at the interface level remain the same, although depending on the instanciation you might have different behaviors in terms of speed for example.

One example of the `List` ADT is given next.

```
import java.util.LinkedList;
import java.util.List;

public class LinkedListExample {

    public static void main(String[] args) {

        List<String> fruits; // declaring a List ADT reference
```

(continues on next page)

(continued from previous page)

```
        fruits = new LinkedList<>(); // Initializing it using LinkedList
        // fruits = new ArrayList<>(); This would also work using ArrayList instead

        // Adding elements
        fruits.add("Apple");
        fruits.add("Banana");
        fruits.add("Cherry");


        // Removing an element
        fruits.remove("Banana");
    }
}
```

In the example above, you see the special notation <> that is associated with *generics* in Java. Generics correspond to the concept of type parameters, allowing you to write code that is parameterized by one or more types. The core idea is to allow type (classes and interfaces) to be parameters when defining classes, interfaces, and methods.

This enables you to create generic algorithms that can work on collections of different types, classes, interfaces, and methods that operate on a parameterized type. Generics offer a way to define and enforce strong type-checks at compile-time without committing to a specific data type.

Java introduced support for generics in 2004, as a part of Java 5 (formally referred to as J2SE 5.0). In earlier versions of Java generics did not exit. You could add any type of object to collections, which was prone to runtime type-casting errors, as illustrated in this example:

```
import java.util.LinkedList;
import java.util.List;

List list = new ArrayList();
list.add("hello");
list.add(1); // This is fine without generics
String s = (String) list.get(1); // ClassCastException at runtime
```

With generics, the type of elements you can add is restricted at compile-time, eliminating the potential for `ClassCastException` at runtime. In the example above, you would have used the `List<String>` and `ArrayList<String>` parametrized classes instead of the `List` and `ArrayList` plain classes. Consequently, the call to `list.add(1)` would have resulted in a compilation error, because 1 is not a `String`.

Generics enable you to write generalized algorithms and classes based on parameterized types, making it possible to reuse the same method, class, or interface for different data types.

4.4.1 Stack ADT

Let us now study in-depth an ADT called `Stack` that is also frequently used by programmers. A stack is a collection that operates on a Last-In-First-Out (LIFO) principle. The primary operations of a `Stack` are `push()`, `pop()`, and `peek()`, as described in the next interface:

```
public interface Stack<T> {
    // Pushes an item onto the top of this stack.
    void push(T item);

```

(continues on next page)

(continued from previous page)

```
    // Removes and returns the top item from this stack.
    T pop();

    // Returns the top item from this stack without removing it.
    T peek();

    // Returns true if this stack is empty.
    boolean isEmpty();

    // Returns the number of items in this stack.
    public int size();
}
```

Let us now see some possible concrete implementations of this interface.

Implementing a Stack With a Linked Structure

The `LinkedStack` is an implementation of the `Stack` ADT that uses a linked list structure to store its elements. Each element in the stack is stored in a node, and each node has a reference to the next node (like individual wagons are connected in a train). The top of the stack is maintained as a reference to the first node (head) of the linked list.

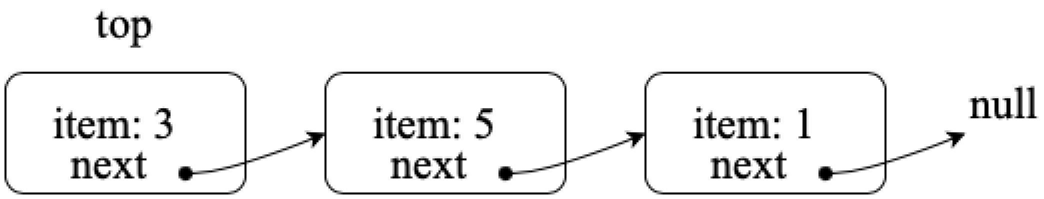
```
public class LinkedStack<T> implements Stack<T> {
    private Node<T> top;
    private int size;

    private static class Node<T> {
        T item;
        Node<T> next;

        Node(T item, Node<T> next) {
            this.item = item;
            this.next = next;
        }
    }

    @Override
    public void push(T item) {
        top = new Node<>(item, top);
        size++;
    }

    @Override
    public T pop() {
        if (isEmpty()) {
            throw new RuntimeException("Stack is empty");
        }
        T item = top.item;
        top = top.next;
        size--;
        return item;
    }
```

(continues on next page)

(continued from previous page)

```
    @Override
    public T peek() {
        if (isEmpty()) {
            throw new RuntimeException("Stack is empty");
        }
        return top.item;
    }

    @Override
    public boolean isEmpty() {
        return top == null;
    }

    @Override
    public int size() {
        return size;
    }
}
```

The state of the linked stack after pushing 1, 5 and 3 in this order is illustrated on the next figure.



Implementing a Stack With an Array

Another method for implementing the Stack ADT is by utilizing an internal array to hold the elements. An implementation is given in the next code fragment:

```
public class DynamicArrayStack<T> implements Stack<T> {
    private T[] array;
    private int top;

    @SuppressWarnings("unchecked")
    public DynamicArrayStack(int initialCapacity) {
        array = (T[]) new Object[initialCapacity];
        top = -1;
    }

    @Override
    public void push(T item) {
        if (top == array.length - 1) {
            resize(2 * array.length); // double the size
        }
        array[++top] = item;
    }
```

(continues on next page)

(continued from previous page)

```java
    @Override
    public T pop() {
        if (isEmpty()) {
            throw new RuntimeException("Stack is empty");
        }
        T item = array[top];
        array[top--] = null; // to prevent memory leak

        // shrink the size if necessary
        if (top > 0 && top == array.length / 4) {
            resize(array.length / 2);
        }
        return item;
    }

    @Override
    public T peek() {
        if (isEmpty()) {
            throw new RuntimeException("Stack is empty");
        }
        return array[top];
    }

    @Override
    public boolean isEmpty() {
        return top == -1;
    }

    @Override
    public int size() {
        return top + 1;
    }

    @SuppressWarnings("unchecked")
    private void resize(int newCapacity) {
        T[] newArray = (T[]) new Object[newCapacity];
        for (int i = 0; i <= top; i++) {
            newArray[i] = array[i];
        }
        array = newArray;
    }
}
```

The internal array is initialized with a size larger than the expected number of elements in the stack to prevent frequent resizing.

An integer variable, often termed `top` or `size`, represents the current position in the stack. When pushing a new element onto the stack, it is added at the position indicated by this integer. Subsequently, the integer is incremented. The `pop()` operation reverses this process: The element at the current position is retrieved, and the integer is decremented. Both the `push()` and `pop()` operations have constant time complexity: $O(1)$.

However, there's an inherent limitation when using arrays in Java: Their size is fixed upon creation. Thus, if the stack's size grows to match the internal array's size, any further push operation risks an

ArrayIndexOutOfBoundsException.

To counteract this limitation, when the internal array is detected to be full, its size is doubled. This is achieved by creating a new array whose capacity is doubled with respect to the current array, then copying the contents of the current array to the new one. Although this resizing operation has a linear time complexity of $O(n)$, where n is the number of elements, it doesn't happen often.

In addition, to avoid inefficiencies in terms of memory usage, if the size of the stack drops to one-quarter of the internal array's capacity, the array size is halved. This prevents the array from being overly sparse and consuming unnecessary memory.

Although resizing (either increasing or decreasing the size) requires $O(n)$ time in the worst case, this cost is distributed over many operations, making the average cost constant. This is known as amortized analysis. Thus, when analyzed in an amortized sense, the average cost per operation over n operations is $O(1)$.

Evaluating Arithmetic Expressions with a Stack

A typical use of stacks is to evaluate arithmetic expressions, as demonstrated in the next algorithm:

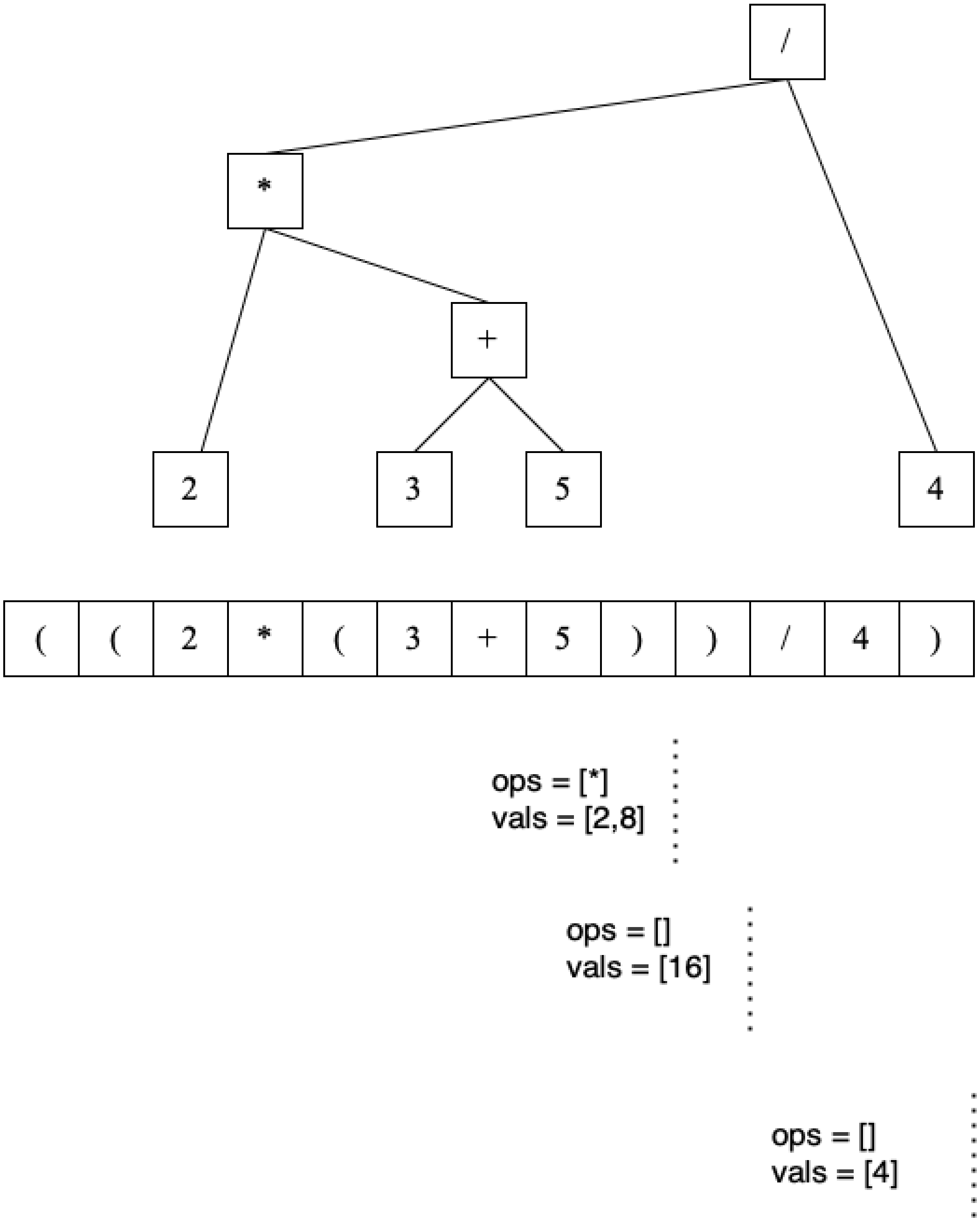
```java
public class ArithmeticExpression {
    public static void main(String[] args) {
        System.out.println(evaluate("( ( 2 * ( 3 + 5 ) ) / 4 )"));
    }

    public static double evaluate(String expression) {

        Stack<String> ops  = new LinkedStack<String>();
        Stack<Double> vals = new LinkedStack<Double>();

        for (String s: expression.split(" ")) {
            // INVARIANT
            if      (s.equals("("))               ;
            else if (s.equals("+"))    ops.push(s);
            else if (s.equals("-"))    ops.push(s);
            else if (s.equals("*"))    ops.push(s);
            else if (s.equals("/"))    ops.push(s);
            else if (s.equals(")")) {
                String op = ops.pop();
                double v = vals.pop();
                if      (op.equals("+"))    v = vals.pop() + v;
                else if (op.equals("-"))    v = vals.pop() - v;
                else if (op.equals("*"))    v = vals.pop() * v;
                else if (op.equals("/"))    v = vals.pop() / v;
                vals.push(v);
            }
            else vals.push(Double.parseDouble(s));
        }
        return vals.pop();

    }
}
```

The time complexity of the algorithm is clearly $O(n)$ where n is the size of the input string:

- Each token (whether it is a number, operator, or parenthesis) in the expression is read and processed exactly once.

- Pushing and popping elements from a stack take constant time, $O(1)$.
- Arithmetic operations (addition, subtraction, multiplication, and division) are performed in constant time, $O(1)$.

To understand and convince one-self about the correctness of the algorithm, we should try to discover an invariant. As can be seen, a fully parenthesized expression can be represented as a binary tree where the parenthesis are not necessary:



(	(	2	*	(	3	+	5	)	)	/	4	)

The internal nodes are the operator and the leaf nodes are the values. The algorithm uses two stacks. One stack (`ops`) is for operators and the other (`vals`) is for (reduced) values. The program splits the input string `args[0]` by spaces to process each token of the expression individually.

We will not formalize completely the invariant here but give some intuition about what it is.

At any point during the processing of the expression:

1. The `vals` stack contains the results of all fully evaluated sub-expressions (reduced subtrees) encountered so far.
2. The `ops` stack contains operators that are awaiting their right-hand operands to form a complete sub-expression (subtree) that can be reduced.
3. For every operator in the `ops` stack, its corresponding left-hand operand is already in the `vals` stack, awaiting the completion of its subtree for reduction.

The figure displays the status of the stacks at three distinct stages for our brief example.

When we encounter an operand, it is like encountering a leaf of this tree, and we immediately know its value, so it is pushed onto the `vals` stack.

When we encounter an operator, it is pushed onto the `ops` stack. This operator is awaiting its right-hand operand to form a complete subtree. Its left-hand operand is already on the `vals` stack.

When a closing parenthesis `)` is encountered, it indicates the end of a fully parenthesized sub-expression, corresponding to an entire subtree of the expression. This subtree is "reduced" or "evaluated" in the following manner:

1. The operator for this subtree is popped from the `ops` stack.
2. The right-hand operand (the value of the right subtree) is popped from the `vals` stack.
3. The left-hand operand (the value of the left subtree) is popped from the `vals` stack.
4. The operator is applied to the two operands, and the result (the value of the entire subtree) is pushed back onto the `vals` stack.

This invariant captures the essence of the algorithm's approach to the problem: It traverses the expression tree in a sort of depth-first manner, evaluating each subtree as it is fully identified by its closing parenthesis.

This algorithm taking a `String` as its input is a an example of an interpreter. Interpreted programming languages (like Python) do similarly but accept constructs that a slightly more complex that parenthesized arithmetic expressions.

Exercise

Write an recursive algorithm for evaluation arithmetic expressions. This program will not use explicit stacks but rely on the call stack instead.

4.4.2 Trees

In many applications, there is a need to store and manage hierarchically organized data. Examples include representing family trees (genealogy), structuring the way files are stored on your computer, or representing arithmetic expressions.

Consider the following example where we represent data using a tree structure.

A tree is often represented with the root node at the top (in this case, the node with the value 5) and the leaf nodes (the ones without children) at the bottom.

The *LinkedBinaryTree* class shown below is an implementation for representing such a tree data-structure. In this class, each node contains a value and references to its left and right children. This structure is similar to the nodes used in linked lists, with the key difference being that we store two references (one for each child) instead of just one. This class facilitates the creation of leaf nodes and includes a static helper method called *combine*, which allows us to merge two trees by creating a new root node, thus forming a larger binary tree.

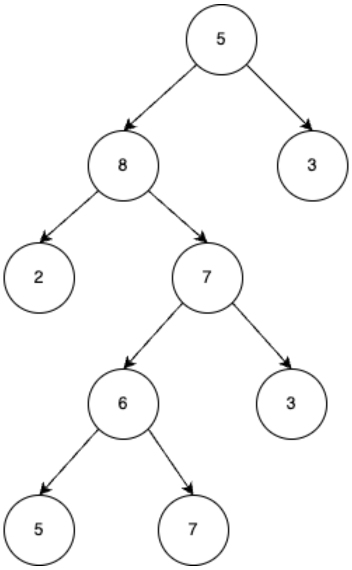


Fig. 2: BinaryTree

```java
public class LinkedBinaryTree {

        private Node root;

        class Node {
            public int val;
            public Node left;
            public Node right;

            public Node(int val) {
                this.val = val;
            }
        }

        public static LinkedBinaryTree leaf(int val) {
            LinkedBinaryTree tree = new LinkedBinaryTree();
            tree.root = tree.new Node(val);
            return tree;
        }

        public static LinkedBinaryTree combine(int val, LinkedBinaryTree left,␣
↪LinkedBinaryTree right) {
            LinkedBinaryTree tree = new LinkedBinaryTree();
            tree.root = tree.new Node(val);
            tree.root.left = left.root;
            tree.root.right = right.root;
            return tree;
        }
}
```

```java
public static void main(String[] args) {
```

(continues on next page)

(continued from previous page)

```
    LinkedBinaryTree tree = combine(5,
                               combine(8,
                                       leaf(2),
                                       combine(7,
                                               combine(6,
                                                       leaf(5),
                                                       leaf(7)),
                                               leaf(3))),
                               leaf(3));
}
```

Tree Traversals

Tree traversal strategies are methods used to visit all the nodes in a tree, such as a binary tree. The three common traversal strategies are pre-order, in-order, and post-order. Here's a brief explanation of each:

- Pre-order traversal visits the current node, then traverse the left subtree, and finally, traverse the right subtree.
- In-order traversal traverses the left subtree, visit the current node, and then traverse the right subtree.
- Post-order Traversal traverses the left subtree, then the right subtree, and finally visit the current node.

The code for each traversal is given next.

```
public void preOrderPrint() {
    preOrderPrint(root);
}

private void preOrderPrint(Node current) {
    if (current == null) {
        return;
    }
    System.out.print(current.val + " ");
    preOrderPrint(current.left);
    preOrderPrint(current.right);
}

public void inOrderPrint() {
    inOrderPrint(root);
}

private void inOrderPrint(Node current) {
    if (current == null) {
        return;
    }
    inOrderPrint(current.left);
    System.out.print(current.val + " ");
    inOrderPrint(current.right);
}

public void postOrderPrint() {
    postOrderPrint(root);
}
```

(continues on next page)

(continued from previous page)

```java
private void postOrderPrint(Node current) {
    if (current == null) {
        return;
    }
    postOrderPrint(current.left);
    postOrderPrint(current.right);
    System.out.print(current.val + " ");
}
```

Here is the output order obtained on the binary represented binary-tree for each traversals:

- Pre-Order: 5, 8, 2, 7, 6, 5, 7, 3, 3
- In-Order: 2, 8, 5, 6, 7, 7, 3, 5, 3
- Post-Order: 2, 5, 7, 6, 3, 7, 8, 3, 5

Visiting a binary tree with n nodes takes $\Theta(n)$ (assuming the visit of one node takes a constant time), since each node is visited exactly once.

Exercise

Write an iterative algorithm (not recursive) for implementing each of these traversals. You will need to use an explicit stack.

We show next two practical examples using binary trees data-structures.

Representing an arithmetic Expression with Tree

The BinaryExpressionTree class in the provided code is an abstract representation of a binary expression tree, a data structure commonly used in computer science for representing expressions with binary operators (like +, -, *, /).

The set of expression methods (mul(), div(), plus(), minus()) allows to build easily expressions from other expressions. These methods return a new OperatorExpressionTree object, which is a subclass of BinaryExpressionTree. Each method takes another BinaryExpressionTree as an operand to the right of the operator. The private static nested class OperatorExpressionTree represents an operator node in the tree with left and right children, which are also BinaryExpressionTree instances. The private static nested class ValueExpressionTree represents a leaf node in the tree that contains a value. A convenience static method value() allows creating a ValueExpressionTree with a given integer value. An example is provided in the main method for creating tree representation of the expression (2 * ((5+7)-3)) / 3.

```java
public abstract class BinaryExpressionTree {


    public BinaryExpressionTree mul(BinaryExpressionTree right) {
        return new OperatorExpressionTree(this, right, '*');
    }

    public BinaryExpressionTree div(BinaryExpressionTree right) {
        return new OperatorExpressionTree(this, right, '/');
    }
```

(continues on next page)

(continued from previous page)

```java
    public BinaryExpressionTree plus(BinaryExpressionTree right) {
        return new OperatorExpressionTree(this, right, '+');
    }

    public BinaryExpressionTree minus(BinaryExpressionTree right) {
        return new OperatorExpressionTree(this, right, '-');
    }

    private static class OperatorExpressionTree extends BinaryExpressionTree {
        private final BinaryExpressionTree left;
        private final BinaryExpressionTree right;
        private final char operator;

        public OperatorExpressionTree(BinaryExpressionTree left, BinaryExpressionTree right, char operator) {
            this.left = left;
            this.right = right;
            this.operator = operator;
        }

    }

    private static class ValueExpressionTree extends BinaryExpressionTree {

        private final int value;

        public ValueExpressionTree(int value) {
            this.value = value;
        }
    }

    public static BinaryExpressionTree value(int value) {
        return new ValueExpressionTree(value);
    }

    public static void main(String[] args) {
        BinaryExpressionTree expr = value(2).mul(value(5).plus(value(7)).minus(value(3)).div(value(3))); // (2 * ((5+7)-3)) / 3
    }

}
```

We now enrich this class with two functionalities:

- `evaluate()` is a method for evaluating the expression represented by the tree. This method performs a post-order traversal of the tree. The evaluation of the left sub-expression (left traversal) and the right subexpression (right traversal) must be first evaluated prior to applying the node operator (visit of the node).
- `prettyPrint()` is a method for printing the expression as full parenthesized representation. It corresponds to an infix traversal. The left subexpression is printed (left traversal) before printing the node operator (visit of the node) and then printing the right subexpression (right traversal).

```java
public abstract class BinaryExpressionTree {

    // evaluate the expression
    abstract int evaluate();

    // print a fully parenthesized representation of the expression
    abstract String prettyPrint();

    // mul , div, plus, minus not represented


    private static class OperatorExpressionTree extends BinaryExpressionTree {
        private final BinaryExpressionTree left;
        private final BinaryExpressionTree right;
        private final char operator;

        // constructor not represented

        @Override
        public String prettyPrint() {
            return "(" + left.prettyPrint() + operator + right.prettyPrint() + ")";
        }

        @Override
        int evaluate() {
            int leftRes = left.evaluate();
            int rightRes = right.evaluate();
            switch (operator) {
                case '+':
                    return leftRes + rightRes;
                case '-':
                    return leftRes - rightRes;
                case '/':
                    return leftRes / rightRes;
                case '*':
                    return leftRes * rightRes;
                default:
                    throw new IllegalArgumentException("unkown operator " + operator);
            }
        }
    }

    private static class ValueExpressionTree extends BinaryExpressionTree {

        private final int value;

        // constructor not represented

        @Override
        public String prettyPrint() {
            return value + "";
        }
```

(continues on next page)

(continued from previous page)

```
        @Override
        int evaluate() {
            return value;
        }
    }
}
```

Exercise

Enrich the BinaryExpressionTree with a method `rpnPrint()` to print the expression in *reverse Polish notation*. In reverse Polish notation, the operators follow their operands. For example, to add 3 and 4 together, the expression is `3 4 +` rather than `3 + 4`. This notation doesn't need parenthesis: `(3 × 4) + (5 × 6)` becomes `3 4 × 5 6 × +` in reverse Polish notation.

Representing a set with a Binary Search Tree (BST)

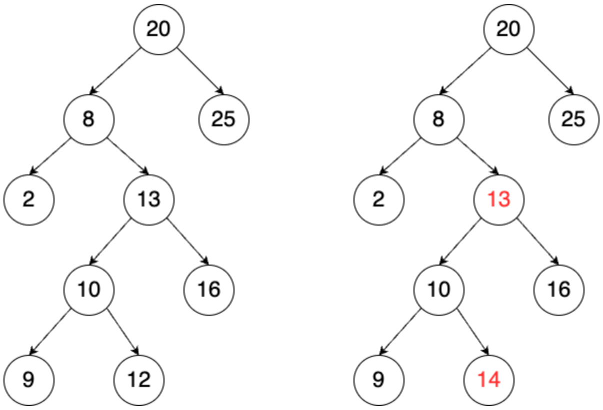
An Abstract Data Type (ADT) for representing a set of integers without duplicates is provided below. The set is initially empty. Elements can then be added, and it is possible to check if an element has already been added.

```
public interface IntSet {
    public void add(int val);
    public boolean contains(int val);
}
```

A Binary Search Tree (BST) is a specialized type of binary tree that can be used as a data-structure for implementing the IntSet ADT. In a BST, each node stores a value, and all nodes in the left subtree of a node contain values less than the node's value, while all nodes in the right subtree contain values greater than the node's value.

These properties ensure that the tree remains ordered, which is crucial for the efficiency of operations like search, insertion, and deletion.

A valid BST is shown on the left, while an invalid BST is displayed on the right. The tree on the right is not valid because the value 14 appears in the left subtree of the node with value 13, which violates the BST rule.



The *BinarySearchTree* class defined below, implements *IntSet*

for storing a set of integers.

```
public class BinarySearchTree implements IntSet {

    private Node root;

    private class Node {
        public int val;
        public Node left;
        public Node right;

        public Node(int val) {
            this.val = val;
        }

    }

    // Method to add a value to the tree
    public void add(int val) {
        root = addRecursive(root, val);
    }

    private Node addRecursive(Node current, int val) {
        if (current == null) {
            return new Node(val);
        }
        if (val < current.val) {
            current.left = addRecursive(current.left, val);
        } else if (val > current.val) {
            current.right = addRecursive(current.right, val);
        } // if val equals current.val, the value already exists, do nothing

        return current;
    }

    // Method to check if the tree contains a specific value
    public boolean contains(int val) {
        return containsRecursive(root, val);
    }

    private boolean containsRecursive(Node current, int val) {
        if (current == null) {
            return false;
        }
        if (val == current.val) {
            return true;
        }
        return val < current.val
                ? containsRecursive(current.left, val)
                : containsRecursive(current.right, val);
    }

    // Main method for testing
    public static void main(String[] args) {
        BinarySearchTree bst = new BinarySearchTree();
```

(continues on next page)

(continued from previous page)

```
        bst.add(5);
        bst.add(3);
        bst.add(7);
        bst.add(1);

        System.out.println("Contains 3: " + bst.contains(3)); // true
        System.out.println("Contains 6: " + bst.contains(6)); // false
    }
}
```

We reuse the linked node representation. The *add(int val)* method allows us to insert a new value into the tree. This method calls *addRecursive(Node current, int val)*, which recursively traverses the tree:

- If the current node is *null*, a new node with the value *val* is created.
- If *val* is less than the current node's value, the algorithm moves (recursively) to the left child.
- If *val* is greater, it moves (recursively) to the right child.
- If *val* is equal to the current node's value, no action is taken (to avoid duplicates).

The recursion ensures that the new value is inserted at the correct position according to the BST properties.

The time complexity for insertion is $\mathcal{O}(h)$, where h is the height of the tree. Since no guarantees on h can be made with this implementation, the height should be assumed to be $\mathcal{O}(n)$, the number of nodes. This worst-case scenario occurs when the tree is highly unbalanced, resembling a linked list. This situation can arise, for instance, if the keys are inserted in increasing order.

Tip: There exist more complex implementations of a BST to enforce h is in $\mathcal{O}(log(n))$. The most famous one is called a red-black tree.

Exercise

One traversal strategy allows to iterate over the values of the BST in increasing order in linear time. Which one is it? Can you implement it?

4.4.3 Maps

An array in Java is a data structure that stores elements in a fixed order. Each element in the array is accessed using an index, which is a integer.

A Map is an ADT that generalizes the idea of indexing to be more flexible by allowing the index to be something else than an integer. In Map, the index is called a key, and each key maps it to a value. An example usage of Map is given next making use of the two most important methods, the *put* to add a key, value pair (aka entry) and the *get* to retrieve the value from the key. Ìn Java, *java.util. `Map* is the interface for the Map ADT. It is implemented by the class *java.util.HashTable* (and many others).

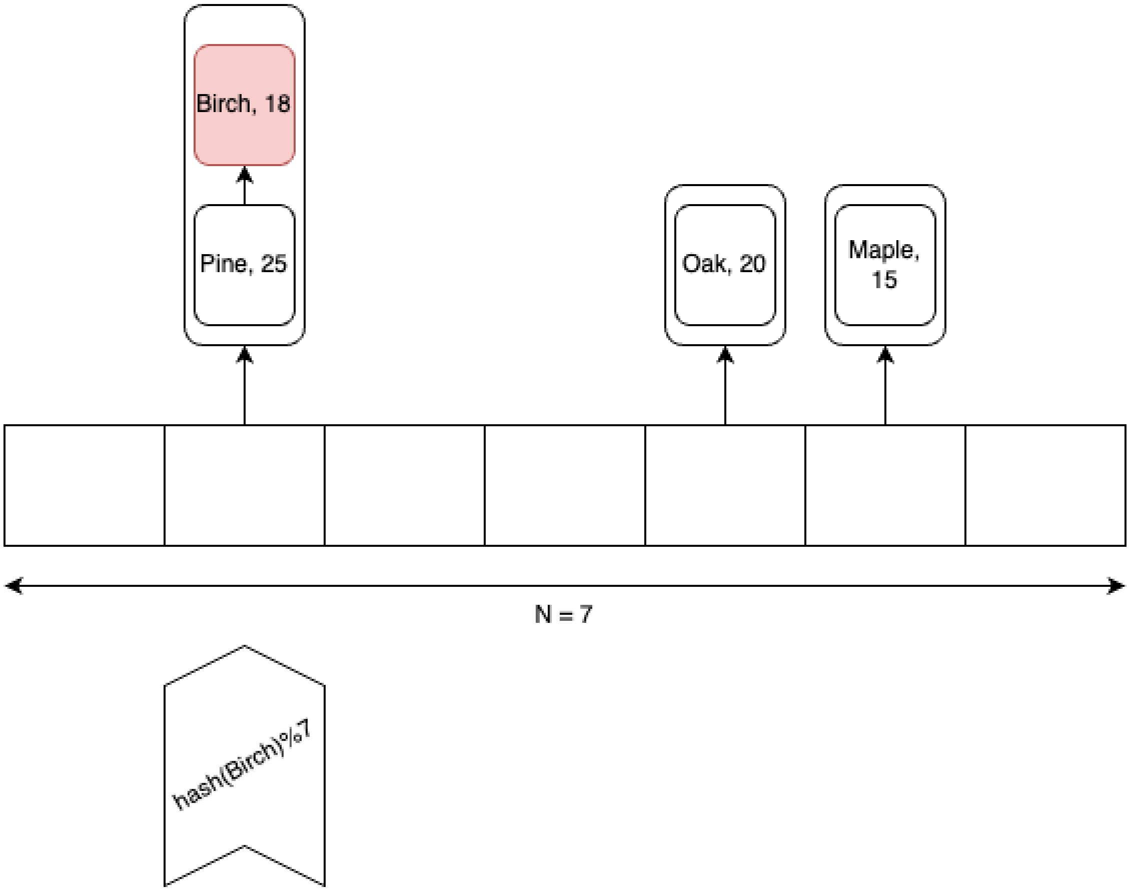
```
import java.util.Hashtable;
import java.util.Map;

public class MapExample {
    public static void main(String[] args) {
```

(continues on next page)

(continued from previous page)

```
        // Create a HashMap to store tree names and their heights
        Map<String, Integer> treeHeights = new HashTable<>();

        // Add some key-value pairs (tree names and their average heights in meters)
        treeHeights.put("Oak", 20);
        treeHeights.put("Pine", 25);
        treeHeights.put("Maple", 15);
        treeHeights.put("Birch", 18);

        // Retrieve the height of a specific tree
        System.out.println("Height of Oak: " + treeHeights.get("Oak")); // Output: 20
    }
}
```

An illustration of the internal representation of hash table is given next.



A hash table stores keys and values as entries (key-value pairs) in a Java array. At the core of a hash table is the concept of hashing. A hash function takes a key (in this case, a string) and converts it into an integer, known as the hash code. It is crucial that this function is deterministic, meaning the hash code for a given key must always remain the same.

The purpose of the hash code is to determine where a specific key should be stored in the internal array. In Java, every object has a *hashCode* method that returns an integer (which can sometimes be negative). To map this hash code to a valid index between 0 and N-1 (where N is the size of the array), we first take its absolute value and then apply the % (modulo) operator.

When inserting a new entry into the hash table, there is no guarantee that the computed index will be unoccupied. This situation, known as a *collision*, occurs when multiple keys are mapped to the same index. To handle collisions, a linked list is used at each index of the array to store multiple entries.

Collisions become more frequent as the number of entries in the hash table grows compared to the size of the array N. The efficiency of the hash table largely depends on the assumption that the hash function distributes keys uniformly across the array and that the number of entries is kept lower than N. However, these topics go beyond the scope of this brief introduction to hash tables. Insertion and retrieval can be assumed to have an expected time complexity of $\mathcal{O}(1)$ for the *java.util.HashTable* implementation because the load factor (ratio of the number of entries divided by N) is kept low by resizing the table if necessary similarly as in the stack implementation with an array.

A simple implementation is given next.

```java
import java.util.LinkedList;

public class SimpleHashtable {

    // Size of the internal array
    private static final int N = 10;

    // Each entry in the array is a LinkedList to handle collisions
    private LinkedList<Entry>[] table;

    // Constructor initializes the array
    @SuppressWarnings("unchecked")
    public SimpleHashtable() {
        table = new LinkedList[N];
        for (int i = 0; i < N; i++) {
            table[i] = new LinkedList<>();
        }
    }

    // A private class representing a key-value pair
    private static class Entry {
        String key;
        Integer value;

        Entry(String key, Integer value) {
            this.key = key;
            this.value = value;
        }
    }

    // Hash function to calculate the index for a given key
    private int hash(String key) {
        return Math.abs(key.hashCode()) % N;
    }

    // Put method to insert or update a key-value pair
    public void put(String key, Integer value) {
        int index = hash(key);
        LinkedList<Entry> bucket = table[index];

        // Check if the key already exists, if so, update its value
```

(continues on next page)

(continued from previous page)

```
        for (Entry entry : bucket) {
            if (entry.key.equals(key)) {
                entry.value = value;
                return;
            }
        }

        // If the key doesn't exist, add a new Entry to the bucket
        bucket.add(new Entry(key, value));
    }

    // Get method to retrieve the value associated with a key
    public Integer get(String key) {
        int index = hash(key);
        LinkedList<Entry> bucket = table[index];

        // Iterate through the bucket to find the key
        for (Entry entry : bucket) {
            if (entry.key.equals(key)) {
                return entry.value;
            }
        }

        // Return null if the key is not found
        return null;
    }
}
```

4.5 Iterators

An iterator is an object that facilitates the traversal of a data structure, especially collections, in a systematic manner without exposing the underlying details of that structure. The primary purpose of an iterator is to allow a programmer to process each element of a collection, one at a time, without needing to understand the inner workings or the specific memory layout of the collection.

Java provides an `Iterator` interface in the `java.util` package, which is implemented by various collection classes. This allows objects of those classes to create iterator instances on demand that can be used to traverse through the collection.

An iterator acts like a cursor pointing to some element within the collection. The two important methods of an iterator are:

- `hasNext()`: Returns `true` if and only if there are more elements to iterate over.
- `next()`: Returns the next element in the collection and advances the iterator. This method fails if `hasNext()` is `false`.

The method `remove()` is optional and will not be covered in this course.

The next example show how to use an iterator to print every element of a list.

```
import java.util.ArrayList;
import java.util.Iterator;
```

(continues on next page)

(continued from previous page)

```java
public class IteratorExample {
    public static void main(String[] args) {
        ArrayList<String> list = new ArrayList<>();
        list.add("A");
        list.add("B");
        list.add("C");

        Iterator<String> it = list.iterator();
        while (it.hasNext()) {
            String element = it.next();
            System.out.println(element);
        }
    }
}
```

`Iterable` should not be confused with `Iterator`. It is also an interface in Java, found in the `java.lang package`. An object is "iterable" if it implements the `Iterable` interface which has a single method: `Iterator<T> iterator();`. This essentially means that the object has the capability to provide an `Iterator` over itself.

Many data structures (like lists, sets, and queues) in the `java.util.collections` package implement the `Iterable` interface to provide a standardized method to iterate over their elements.

One of the main benefits of the `Iterable` interface is that it allows objects to be used with the *enhanced for-each loop* in Java. Any class that implements `Iterable` can be used in a for-each loop. This is illustrated next that is equivalent to the previous code.

```java
import java.util.ArrayList;
import java.util.Iterator;

public class IteratorExample {
    public static void main(String[] args) {
        ArrayList<String> list = new ArrayList<>();
        list.add("A");
        list.add("B");
        list.add("C");

        for (String element: list) {
            System.out.println(element);
        }
    }
}
```

In conclusion, while they are closely related and often used together, `Iterable` and `Iterator` serve distinct purposes. `Iterable` is about the ability to produce an `Iterator`, while `Iterator` is the mechanism that actually facilitates the traversal.

4.5.1 Implementing your own iterators

The correct way to enable iteration capabilities is to extend the Iterable interface at the Abstract Data Type (ADT) level. This ensures that any implementation of the ADT is required to implement the iterator method defined by the Iterable interface.

```java
public interface Stack<T> extends Iterable<T> {
    // Pushes an item onto the top of this stack.
    void push(T item);

    // Removes and returns the top item from this stack.
    T pop();

    // Returns the top item from this stack without removing it.
    T peek();

    // Returns true if this stack is empty.
    boolean isEmpty();

    // Returns the number of items in this stack.
    public int size();
}
```

Now to properly implement an `Iterator`, there are two possible strategies:

1. Fail-Fast: Such iterators throw `ConcurrentModificationException` if there is structural modification of the collection.
2. Fail-Safe: Such iterators don't throw any exceptions if a collection is structurally modified while iterating over it. This is because they operate on the clone of the collection, not on the original collection.

Fail-Safe iterator may be slower since one have to pay the cost of the clone at the creation of the iterator, even if we only end-up iterating over few elements. Therefore we will rather focus on the Fail-Fast strategy, which corresponds to the most frequent choice in the implementation of Java collections.

To implement a Fail-Fast iterator for our `LinkedStack`, we can keep track of a modification count for the stack. This count will be incremented whenever there's a structural modification to the stack (like pushing or popping). The iterator will then capture this count when it is created and compare its own captured count to the stack's modification count during iteration. If they differ, the iterator will throw a `ConcurrentModificationException`. The `LinkedStack` class has an inner `LinkedStackIterator` class that checks the modification count every time it is asked if there's a next item or when retrieving the next item. It is important to understand that `LinkedStackIterator` is an inner class, *not* a static nested class. An inner class cannot be instantiated without first instantiating the outer class and it is tied to a specific instance of the outer class. This is why, the instance variables of the `Iterator` inner class can be initialized using the instance variables of the outer class.

The sample main method demonstrates that trying to modify the stack during iteration (by pushing a new item) results in a `ConcurrentModificationException`.

The creation of the iterator has a constant time complexity, $O(1)$. Indeed:

1. The iterator's current node is set to the top node of the stack. This operation is done in constant time since it is just a reference assignment.
2. Modification Count Assignment: The iterator captures the current modification count of the stack. This again is a simple assignment operation, done in constant time.

No other operations are involved in the iterator's creation, and notably, there are no loops or recursive calls that would add to the time complexity. Therefore, the total time complexity of creating the LinkedStackIterator is $O(1)$.

```java
import java.util.Iterator;
import java.util.ConcurrentModificationException;

public class LinkedStack<T> implements Stack<T> {
    private Node<T> top;
    private int size = 0;
    private int modCount = 0;  // Modification count

    private static class Node<T> {
        private T item;
        private Node<T> next;

        Node(T item, Node<T> next) {
            this.item = item;
            this.next = next;
        }
    }

    public void push(T item) {
        Node<T> oldTop = top;
        top = new Node<>(item, oldTop);
        size++;
        modCount++;
    }

    public T pop() {
        if (top == null) throw new IllegalStateException("Stack is empty");
        T item = top.item;
        top = top.next;
        size--;
        modCount++;
        return item;
    }

    public boolean isEmpty() {
        return top == null;
    }

    public int size() {
        return size;
    }

    @Override
    public Iterator<T> iterator() {
        return new LinkedStackIterator();
    }

    private class LinkedStackIterator implements Iterator<T> {
        private Node<T> current = top;
        private final int expectedModCount = modCount;

        @Override
        public boolean hasNext() {
```

(continues on next page)

(continued from previous page)

```
            if (expectedModCount != modCount) {
                throw new ConcurrentModificationException();
            }
            return current != null;
        }

        @Override
        public T next() {
            if (expectedModCount != modCount) {
                throw new ConcurrentModificationException();
            }
            if (current == null) throw new IllegalStateException("No more items");

            T item = current.item;
            current = current.next;
            return item;
        }
    }

    public static void main(String[] args) {
        Stack<Integer> stack = new LinkedStack<>();
        stack.push(1);
        stack.push(2);
        stack.push(3);

        Iterator<Integer> iterator = stack.iterator();
        while (iterator.hasNext()) {
            System.out.println(iterator.next());
            stack.push(4);  // Will cause ConcurrentModificationException at the next␣
↪call to hasNext
        }
    }
}
```

CHAPTER

FIVE

PARALLEL PROGRAMMING

Parallel programming is a computing technique where multiple tasks are executed simultaneously to achieve faster execution times and to improve resource utilization, compared to sequential execution.

Parallel programming involves **breaking down a task into smaller sub-tasks that can be executed independently and concurrently** across multiple processors, such as CPUs (central processing units), GPUs (graphics processing units), or distributed computing resources.

Parallel programming is often introduced as a way to optimally take advantage of the multiple computing units that are embedded in modern computers, in order to **speed up some computation that takes a lot time**.

However, besides this exploitation of multiple computing units to speed up computations, parallel programming also enables the design of **responsive user interfaces**. Indeed, most GUI (graphical user interface) frameworks are built on the top of a “main event loop” that continuously monitors user interactions and that calls application code to react to those events. If the application code takes too much time to run, the user interface appears to “freeze.” Parallel programming allows to run this applicative logic in the background, hereby preserving an optimal user experience.

This concept of “main event loop” can also be encountered in **network applications**, where a software must simultaneously serve requests issued by different clients. Thanks to parallel programming, each connection with a client can be handled in the background, leaving the main server able to listen to new connections.

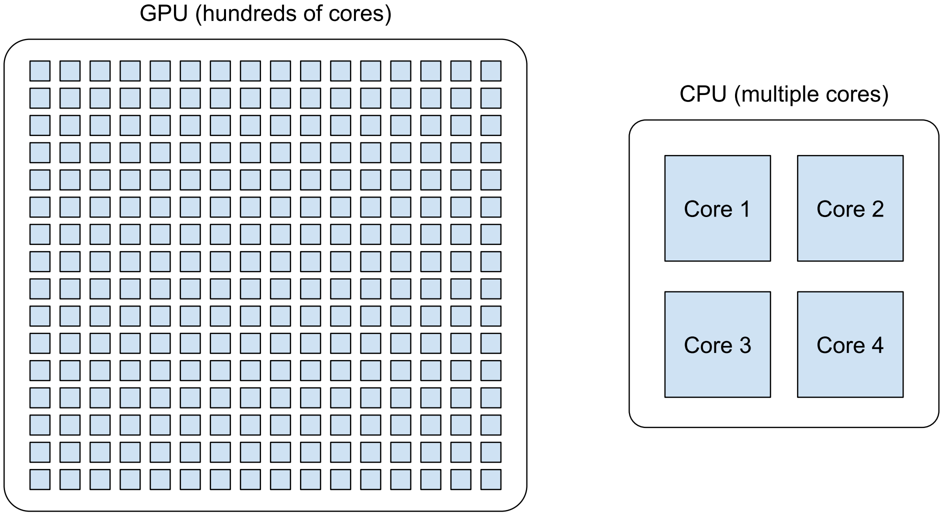
Finally, it may also happen that the design of a whole software can be more naturally modeled using parallel programming than using sequential programming. Think about your personal week agenda: You have a number of distinct tasks of different natures to be achieved during the week, and those tasks only have loose dependencies between them. A large-scale software is likewise: It can generally be **decomposed into a set of mostly uncoupled tasks**, where each of the individual tasks has a different objective and can be solved using sequential programming. Sequential programming has indeed the advantage of being easier to write, to *test*, and to correct. Nevertheless, developing the whole software using sequential programming would introduce unnecessary, arbitrary dependencies between the individual tasks, hereby reducing the performance and increasing the complexity of the design. Parallel programming can be a solution to improve such designs.

5.1 GPUs vs. CPUs

In recent years, there is a growing interest in the exploitation of GPUs to carry on computations that are not related to computer graphics. Indeed, **GPUs** consist of a massive number of processing units working in parallel that are highly optimized for certain types of computations, especially those involving graphics rendering, heavy matrix operations, numerical simulations, and deep learning.

However, while GPUs are incredibly powerful for certain types of parallel computations, they are not a universal replacement for CPUs. Indeed, CPUs are much more versatile, as they are designed to handle a wide range of tasks, including general-purpose computing, running operating systems, managing I/O operations, executing single-threaded applications, and handling diverse workloads. In contrast, the processing units of GPUs focus on simpler, identical tasks that can be duplicated a large number of times. Furthermore, certain types of tasks, particularly those with sequential dependencies or requiring frequent access to shared data, might not benefit significantly from GPU acceleration. Finally, writing code for GPUs often requires the usage of specialized programming languages or libraries and the understanding of the underlying hardware architecture.

Consequently, in this course, we will only focus on **parallel programming on CPUs**. It is indeed essential to notice that thanks to hardware evolution, any consumer computer is nowadays equipped with multiple CPU cores (even if they are less numerous than processing units inside a GPU), as depicted on the following picture:



Parallel programming on CPU seeks to leverage the multiple CPU cores available inside a single computer to execute multiple tasks or portions of a single task simultaneously.

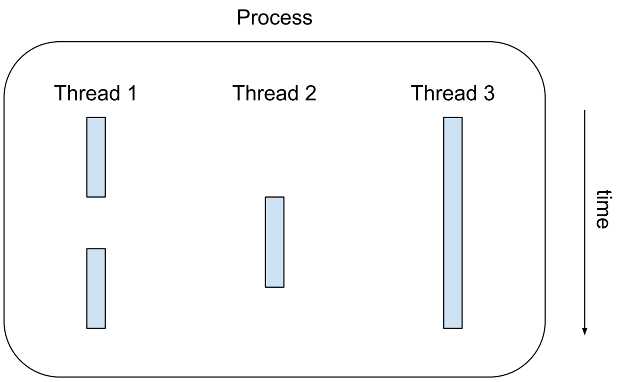
5.2 Multiprocessing vs. multithreading

In computing, a **process** corresponds to a program that is actively running on the CPU of a computer, along with its current state. A typical operating system allows multiple independent processes to run concurrently on the available CPU cores, hereby providing an environment to achieve parallelism that is referred to as **multiprocessing**.

A process has its own memory space (including code, data, stack, and CPU registers) and its own resources that are isolated from other processes to prevent unauthorized access and interference. Distinct processes may still communicate with each other through the so-called **"interprocess communication" (IPC)** mechanisms provided by the operating system. IPCs include files, pipes, message passing, shared memory, and network communications (sockets).

Multiprocessing has two main downsides. Firstly, creating new processes incurs a high overhead due to the need for separate memory allocation and setup for each process. Secondly, because the different processes are isolated from each other, interprocess communications are relatively complex and come at a non-negligible cost.

This motivates the introduction of the concept of a **thread**. A thread refers to the smallest unit of execution within a process: A thread corresponds to a sequence of instructions that can be scheduled and executed independently by one CPU core. One single process can run multiple threads, as illustrated below:



In this picture, the blue blocks indicate at which moment the different threads are active (i.e., are executing something) within the process. A thread can indeed "fall asleep" while waiting for additional data to process, while waiting for user interaction, or while waiting for the result of a computation done by another thread.

Accordingly, **multithreading** is a programming technique where a single process is divided into multiple threads of execution. Threads can perform different operations concurrently, such as doing a computation in the background or handling different parts of the application (e.g., keeping the user interface responsive or serving requests from multiple clients).

Importantly, contrarily to processes, **threads within the same process are not isolated**: They share the same memory space and resources, which allows distinct threads to directly access the same variables and data structures. Threads are sometimes called *lightweight processes*, because creating threads within a process incurs less overhead compared to creating separate processes.

Summarizing, multithreading tends to be simpler and more lightweight than multiprocessing. This explains why this course will only cover the **basics of multithreading in Java**.

It is always worth remembering that the fact that different threads do not live in isolation can be error-prone. Multithreading notably requires the introduction of suitable synchronization and coordination mechanisms between threads when accessing shared variables. If not properly implemented, **race conditions, deadlocks, and synchronization issues can emerge**, which can be extremely hard to identify and resolve.

Also, note that all programmers are constantly confronted with threads. Indeed, even if you never explicitly create a thread by yourself, the vast majority of software frameworks (such as GUI frameworks and a software libraries that deal with network programming or scientific computations) will create threads on your behalf. For instance, in the context of Java-based GUI, both the *AWT (Abstract Window Toolkit) and the Swing frameworks* will transparently create threads to handle the interactions with the user. Consequently, parallel programming should never be considered as an "advanced feature" of a programming language, because almost any software development has to deal with threads. In other words, even if you do not create your own threads, it is important to understand how to design thread-safe applications that properly coordinate the accesses to the shared memory space.

5.3 Threads in Java

Java provides extensive support of multithreading.

When a Java program starts its execution, the Java Virtual Machine (JVM) starts an initial thread. This initial thread is called the **main thread** and is responsible for the execution of the `main()` method, which is the *entry point of most Java applications*. Alongside the main thread, the JVM also start some private background threads for its own housekeeping (most notably the garbage collector).

Additional threads can then be created by software developers in two different ways:

- By *extending* the standard class Thread. Note that since Thread belongs to the java.lang package, no import directive is needed. Here is the documentation of the Thread class: https://docs.oracle.com/javase/8/docs/api/java/lang/Thread.html
- By *implementing* the standard interface Runnable that is also part of the java.lang package: https://docs.oracle.com/javase/8/docs/api/java/lang/Runnable.html

In this course, we will use the second approach. The Runnable interface is quite intuitive:

```java
public interface Runnable {
    public void run();
}
```

This snippet indicates that to create a thread, we first have to define a class providing a run() method that will take care of the computations. Once a concrete class implementing this Runnable interface is available, it can be executed as a thread by instantiating an object of the Thread class.

5.3.1 Using a thread to compute the minimum

As an illustration, let us consider the task of computing the minimum value of an array of floating-point numbers. It is straightforward to implement a sequential method to do this computation:

```java
static public void computeMinValue(float values[]) {
    if (values.length == 0) {
        System.out.println("This is an empty array");
    } else {
        float minValue = values[0];
        for (int i = 1; i < values.length; i++) {
            if (values[i] < minValue) {
                minValue = values[i];
            }
            // One could have written more compactly: minValue = Math.min(minValue,␣
↪values[i]);
        }
        System.out.println("Minimum value: " + minValue);
    }
}
```

As explained above, if one wishes to run this computation as a background thread, the computeMinValue() method must wrapped inside some implementation of the Runnable interface. But the run() method of the Runnable interface does not accept any parameter, so we cannot directly give the values array as an argument to run(). The trick is to store a reference to the values array inside the class that implements Runnable:

```java
class MinComputation implements Runnable {
    private float values[];

    public MinComputation(float values[]) {
        this.values = values;
    }

    @Override
    public void run() {
        computeMinValue(values);
```

(continues on next page)

(continued from previous page)

```
    }
}
```

Our MinComputation class specifies how to compute the minimum of an array. We can evidently run this computation in a purely sequential way as follows:

```
public static void main(String[] args) {
    float values[] = new float[] { -2, -5, 4 };
    Runnable r = new MinComputation(values);
    r.run();
    // This prints: "Minimum value: -5.0"
}
```

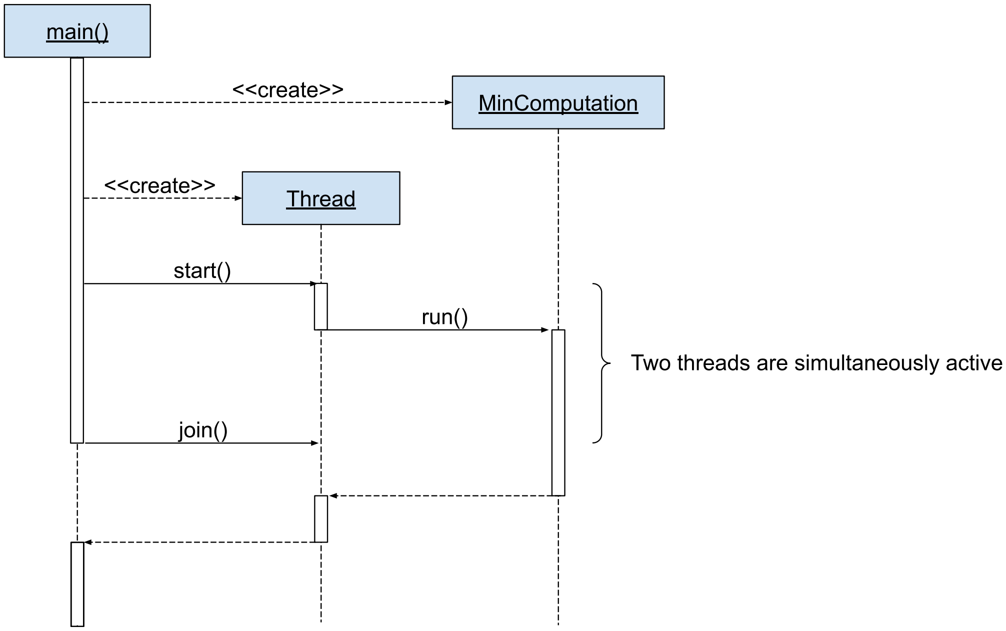
In this example, no additional thread was created besides the main Java thread. Thanks to the fact that MinComputation implements Runnable, it is now possible to compute the minimum in a separate thread:

```
public static void main(String[] args) {
    float values[] = new float[] { -2, -5, 4 };

    // First, create a thread that specifies the computation to be done
    Thread t = new Thread(new MinComputation(values));

    // Secondly, start the thread
    t.start();

    // ...at this point, the main thread can do other stuff...
    System.out.println("This is the main thread");

    // Thirdly, wait for the thread to complete its computation
    try {
        t.join();
    } catch (InterruptedException e) {
        throw new RuntimeException("Unexpected interrupt", e);
    }

    System.out.println("All threads have finished");
}
```

As can be seen in this example, doing a computation in a background thread involves three main steps:

1. Construct an object of the class Thread out of an object that implements the Runnable interface.
2. Launch the thread by using the start() method of Thread. The constructor of Thread does not automatically start the thread, so we have to do this manually.
3. Wait for the completion of the thread by calling the join() method of Thread. Note that join() can throw an InterruptedException, which happens if the thread is interrupted by something.

The following sequence diagram (loosely inspired from UML) depicts this sequence of calls:

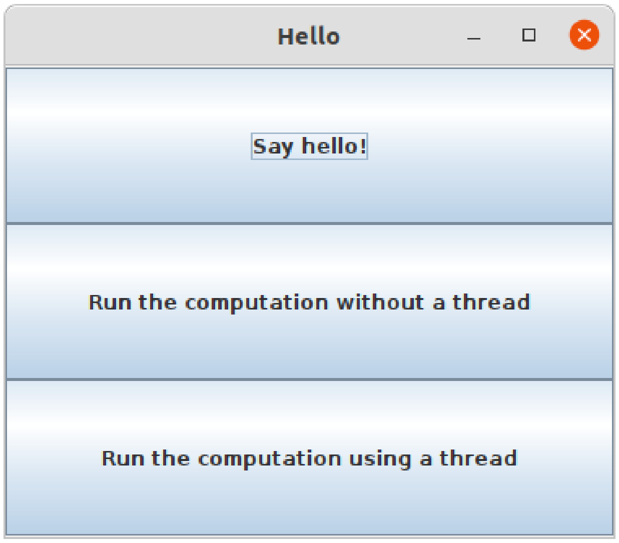


In this diagram, the white bands indicate the moments where the different objects are executing code. It can be seen that between the two calls `t.start()` and `t.join()`, two threads are simultaneously active: the main thread and the computation thread. Note that once the main thread calls `t.join()`, it falls asleep until the computation thread finishes its work.

In other words, the `t.join()` call is a form of **synchronization** between threads. It is always a good idea for the main Java thread to wait for all of its child threads by calling `join()` on each of them. If a child thread launches its own set of sub-threads, it is highly advised for this child thread to call `join()` of each of its sub-threads before ending. The Java process will end if all its threads have ended, including the main thread.

5.3.2 Keeping user interfaces responsive

The `MinComputation` example creates one background thread to run a computation on a array. As explained in the *introduction*, this software architecture can have an interest to keep the user interface responsive during some long computation. To illustrate this interest, consider a *GUI application* with three buttons using Swing:


Hello
Say hello!
Run the computation without a thread
Run the computation using a thread

The corresponding source code is:

```java
import java.awt.GridLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JOptionPane;
import javax.swing.JPanel;

class SayHello implements ActionListener {
    @Override
    public void actionPerformed(ActionEvent e) {
        JOptionPane.showMessageDialog(null, "Hello world!");
    }
}

public class ButtonThread {
    public static void main(String[] args) {
        JFrame frame = new JFrame("Hello");
        frame.setSize(400,200);
        frame.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);

        JPanel panel = new JPanel();
        panel.setLayout(new GridLayout(3, 1));
        frame.add(panel);

        JButton button1 = new JButton("Say hello!");
        button1.addActionListener(new SayHello());
        panel.add(button1);

        JButton button2 = new JButton("Run the computation without a thread");
        button2.addActionListener(new RunWithoutThread());
        panel.add(button2);

        JButton button3 = new JButton("Run the computation using a thread");
        button3.addActionListener(new RunUsingThread());
        panel.add(button3);

        frame.setVisible(true);
    }
}
```

Once the user clicks on the “Say hello!” button, a message box appears saying “Hello world!”. Let us now implement the button entitled “Run the computation without a thread”. In the `ActionListener` observer associated with this button, a long computation is emulated by waiting for 3 seconds:

```java
static public void expensiveComputation() {
    try {
        Thread.sleep(3000);
    } catch (InterruptedException e) {
    }
    JOptionPane.showMessageDialog(null, "Phew! I've finished my hard computation!");
}
```

(continues on next page)

(continued from previous page)

```
class RunWithoutThread implements ActionListener {
    @Override
    public void actionPerformed(ActionEvent e) {
        expensiveComputation();
    }
}
```

If you try and run this example, if clicking on this second button, it becomes impossible to do any other interaction with the "Say hello!" button. The user interface is totally frozen until the `expensiveComputation()` method finishes its work.

In order to turn this non-responsive application into a responsive application, one can simply start a thread that runs the `expensiveComputation()` method:

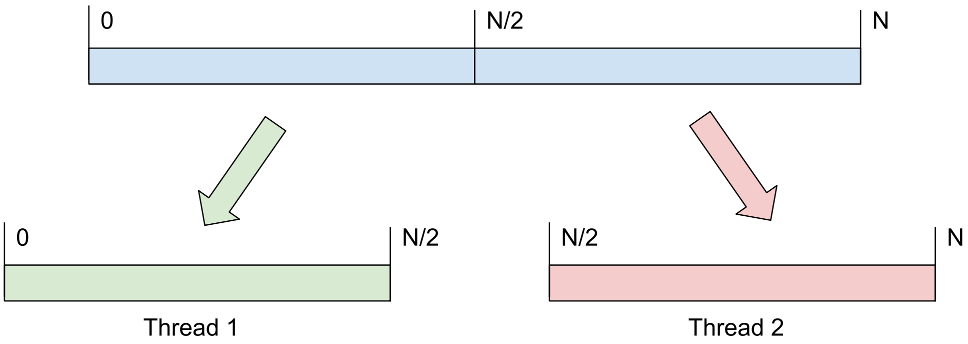
```
class Computation implements Runnable {
    @Override
    public void run() {
        expensiveComputation();
    }
}

class RunUsingThread implements ActionListener {
    @Override
    public void actionPerformed(ActionEvent e) {
        Thread t = new Thread(new Computation());
        t.start();
    }
}
```

This way, even when the computation is running, it is still possible to click on "Say hello!".

5.3.3 Speeding up the computation

Even though starting a background thread can be interesting to improve the responsiveness of an application (*as illustrated above*), this does not speed up the computation. For instance, the time that is necessary to *compute the minimum value* using `MinComputation` is still the same as the purely sequential implementation of method `computeMinValue()`. In order to reduce the computation time, it is needed to modify the sequential algorithm so that it can exploit multiple CPU cores.

For algorithms working on an array, the basic idea is to split the array in two parts, then to process each of those parts by two distinct threads:



Once the two threads have finished their work, we need to **combine** their results to get the final result. In our example, the minimum of the whole array is the minimum of the two minimums computed on the two parts.

To implement this solution, the class that implements the `Runnable` interface must not only receive the `values` array, but it must also receive the start index and the end index of the block of interest in the array. Furthermore, the class must not *print* the minimum, but must provide access to computed minimum value in either block. This is implemented in the following code:

```java
class MinBlockComputation implements Runnable {
    private float values[];
    private int startIndex;
    private int endIndex;
    private float minValue;

    public MinBlockComputation(float values[],
                               int startIndex,
                               int endIndex) {
        if (startIndex >= endIndex) {
            throw new IllegalArgumentException("Empty array");
        }

        this.values = values;
        this.startIndex = startIndex;
        this.endIndex = endIndex;
    }

    @Override
    public void run() {
        minValue = values[startIndex];
        for (int i = startIndex + 1; i < endIndex; i++) {
            minValue = Math.min(values[i], minValue);
        }
    }

    float getMinValue() {
        return minValue;
    }
}
```

Note that we now have to throw an exception if the array is empty, because the minimum is not defined in this case. In the previous implementation, we simply printed out the information. This is not an appropriate solution anymore, as we have to provide an access to the computed minimum value.

Thanks to this new design, it is now possible to speed up the computation the minimum using two threads:

```
public static void main(String[] args) throws InterruptedException {
    float values[] = new float[] { -2, -5, 4 };

    MinBlockComputation c1 = new MinBlockComputation(values, 0, values.length / 2);
    MinBlockComputation c2 = new MinBlockComputation(values, values.length / 2, values.
↪length);

    Thread t1 = new Thread(c1);
    Thread t2 = new Thread(c2);

    t1.start();
    t2.start();

    t1.join();
    t2.join();

    System.out.println("Minimum is: " + Math.min(c1.getMinValue(), c2.getMinValue()));
}
```

The implementation works as follows:

1. We define the two computations `c1` and `c2` that must be carried on the two parts of the whole array. Importantly, the computations are only *defined*, the minimum is not computed at this point.
2. We create and launch two threads `t1` and `t2` that will respectively be in charge of calling the `c1.run()` and `c2.run()` methods. In other words, it is only *after* the calls to `t1.start()` and `t2.start()` that the search for the minimum begins.
3. Once the two threads have finished their work, the main thread collects the partial results from `c1` and `c2`, then combines these partial results in order to print the final result.

Also note that this version does not catch the possible `InterruptedException`, but reports it to the caller.

5.3.4 Dealing with empty parts

Even though the implementation from the previous section works fine on arrays containing at least 2 elements, it fails if the `values` array is empty or only contains 1 element. Indeed, in this case, `values.length / 2 == 0`, which throws the `IllegalArgumentException` in the constructor of `c1`. Furthermore, if `values.length == 0`, the constructor of `c2` would launch the same exception.

One could solve this problem by conditioning the creation of `c1`, `c2`, `t1`, and `t2` according to the value of `values.length`. This would however necessitate to deal with multiple cases that are difficult to write and maintain. This problem would also be exacerbated if we decide to divide the array into more than 2 parts to better exploit the available CPU cores.

A simpler, more scalable solution consists in introducing a Boolean flag that indicates whether a result is present for each part of the array. Instead of throwing the `IllegalArgumentException` in the constructor, this flag would be set to `false` if the search for minimum is launched on an empty block.

To illustrate this idea, let us consider the slightly more complex problem of computing both the minimum and the maximum values of an array. The first step is to define a class that will hold the result of a computation:

```
class MinMaxResult {
    private boolean isPresent;
```

(continues on next page)

(continued from previous page)

```
    private float minValue;
    private float maxValue;

    private MinMaxResult(boolean isPresent,
                         float minValue,
                         float maxValue) {
        this.isPresent = isPresent;
        this.minValue = minValue;
        this.maxValue = maxValue;
    }

    public MinMaxResult(float minValue,
                        float maxValue) {
        this(true /* present */, minValue, maxValue);
    }

    static public MinMaxResult empty() {
        return new MinMaxResult(false /* not present */, 0 /* dummy min */, 0 /* dummy max */);
    }

    public boolean isPresent() {
        return isPresent;
    }

    public float getMinValue() {
        if (isPresent()) {
            return minValue;
        } else {
            throw new IllegalStateException();
        }
    }

    public float getMaxValue() {
        if (isPresent()) {
            return maxValue;
        } else {
            throw new IllegalStateException();
        }
    }

    public void print() {
        if (isPresent()) {
            System.out.println(getMinValue() + " " + getMaxValue());
        } else {
            System.out.println("Empty array");
        }
    }

    public void combine(MinMaxResult with) {
        if (with.isPresent) {
            if (isPresent) {
```

(continues on next page)

(continued from previous page)

```
                // Combine the results from two non-empty blocks
                minValue = Math.min(minValue, with.minValue);
                maxValue = Math.max(maxValue, with.maxValue);
            } else {
                // Replace the currently absent result by the provided result
                isPresent = true;
                minValue = with.minValue;
                maxValue = with.maxValue;
            }
        } else {
            // Do nothing if the other result is absent
        }
    }
}
```

Introducing the `MinMaxResult` class allows us to cleanly separate the two distinct concepts of the "*algorithm to do a computation*" and of the "*results of the computation*." This separation is another example of a *design pattern*.

As can be seen in the source code, there are two possible ways to create an object of the `MinMaxResult` class:

- either by using the `MinMaxResult(minValue, maxValue)` constructor, which sets the `isPresent` flag to `true` in order to indicate the presence of a result,
- or by using the `MinMaxResult.empty()` static method, that creates a `MinMaxResult` object with the `isPresent` flag set to `false` in order to indicate the absence of a result (this is the case of an empty block).

The object throws an exception if trying to access the minimum or the maximum values if the result is absent. It is up to the caller to check the presence of a result using the `isPresent()` method, before calling `getMinValue()` or `getMaxValue()`.

Finally, note the presence of the `combine()` method. This method updates the currently available minimum/maximum values with the results obtained from a different block. The `combine()` implements the combination of two partial results.

It is now possible to create an implementation of the `Runnable` interface that leverages `MinMaxResult`:

```
class MinMaxBlockComputation implements Runnable {
    private float[] values;
    private int startIndex;
    private int endIndex;
    private MinMaxResult result;

    public MinMaxBlockComputation(float[] values,
                                  int startIndex,
                                  int endIndex) {
        this.values = values;
        this.startIndex = startIndex;
        this.endIndex = endIndex;
    }

    @Override
    public void run() {
        if (startIndex >= endIndex) {
            result = MinMaxResult.empty();
        } else {
```

(continues on next page)

(continued from previous page)

```
            float minValue = values[startIndex];
            float maxValue = values[startIndex];

            for (int i = startIndex + 1; i < endIndex; i++) {
                minValue = Math.min(minValue, values[i]);
                maxValue = Math.max(maxValue, values[i]);
            }

            result = new MinMaxResult(minValue, maxValue);
        }
    }

    MinMaxResult getResult() {
        return result;
    }
}
```

The MinMaxBlockComputation class is essentially the same as the MinBlockComputation class *defined earlier*. It only differs in the way the result is stored: MinBlockComputation uses a float to hold the result of the computation on a block, whereas MinMaxBlockComputation uses an object of the MinMaxResult class. This allows MinMaxBlockComputation not only to report both the minimum and maximum values of part of an array, but also to indicate whether that part was empty or non-empty.

It is now easy to run the computation using two threads in a way that is also correct when the values array contains 0 or 1 element:

```
public static void main(String[] args) throws InterruptedException {
    float values[] = new float[1024];
    // Fill the array

    MinMaxBlockComputation c1 = new MinMaxBlockComputation(values, 0, values.length / 2);
    MinMaxBlockComputation c2 = new MinMaxBlockComputation(values, values.length / 2,␣
↪values.length);
    Thread t1 = new Thread(c1);
    Thread t2 = new Thread(c2);
    t1.start();
    t2.start();
    t1.join();
    t2.join();

    MinMaxResult result = c1.getResult();
    result.combine(c2.getResult());
    result.print();
}
```

5.3.5 Optional results

The MinMaxResult class *was previously introduced* as a way to deal with the absence of a result in the case of an empty part of an array. More generally, dealing with the absence of a value is a common pattern in software architectures. For this reason, Java introduces the Optional<T> generic class: https://docs.oracle.com/javase/8/docs/api/java/util/Optional.html

The Optional<T> class does exactly the same stuff as the isPresent Boolean flag that we manually introduced into the MinMaxResult class. The four main operations of Optional<T> are:

- of(T t) is a static method that constructs an Optional<T> object embedding the given object t of class T.
- empty() is a static method that constructs an Optional<T> object indicating the absence of an object of class T.
- isPresent() is a method that indicates whether the Optional<T> object contains an object.
- get() returns the embedded object of class T. If the Optional<T> does not contains an object, an exception is thrown.

Consequently, we could have defined a simplified version of MinMaxResult without the isPresent Boolean flag as follows:

```java
class MinMaxResult2 {
    private float minValue;
    private float maxValue;

    public MinMaxResult2(float minValue,
                         float maxValue) {
        this.minValue = minValue;
        this.maxValue = maxValue;
    }

    public float getMinValue() {
        return minValue;
    }

    public float getMaxValue() {
        return maxValue;
    }
}
```

By combining MinMaxResult2 with Optional<T>, the sequential algorithm to be integrated inside the run() method of the Runnable class could have been rewritten as:

```java
import java.util.Optional;

public static Optional<MinMaxResult2> computeMinMaxSequential(float values[],
                                                              int startIndex,
                                                              int stopIndex) {
    if (startIndex >= stopIndex) {
        return Optional.empty();
    } else {
        float minValue = values[startIndex];
        float maxValue = values[startIndex];
```

(continues on next page)

(continued from previous page)

```
        for (int i = startIndex + 1; i < stopIndex; i++) {
            minValue = Math.min(minValue, values[i]);
            maxValue = Math.max(maxValue, values[i]);
        }

        return Optional.of(new MinMaxResult2(minValue, maxValue));
    }
}

public static void main(String[] args) {
    float values[] = new float[] { -2, -5, 4 };

    Optional<MinMaxResult2> result = computeMinMaxSequential(values, 0, values.length);
    if (result.isPresent()) {
        System.out.println(result.get().getMinValue() + " " + result.get().
↪getMaxValue());
    } else {
        System.out.println("Empty array");
    }
}
```

This alternative implementation would have been slightly shorter and would have avoided any possible bug in our manual implementation of the `isPresent` flag.

Exercise

Reimplement the `MinMaxBlockComputation` class by replacing `MinMaxResult` with `Optional<MinMaxResult2>`, and launch threads based on this new class.

5.4 Thread pools

So far, we have only created two threads, but a modern CPU will typically have at least 4 cores. One could launch more threads to benefit from those additional cores. For instance, the following code would use 4 threads by dividing the array in 4 parts:

```
public static void main(String[] args) throws InterruptedException {
    float values[] = new float[1024];
    // Fill the array

    int blockSize = values.length / 4;
    MinMaxBlockComputation c1 = new MinMaxBlockComputation(values, 0, blockSize);
    MinMaxBlockComputation c2 = new MinMaxBlockComputation(values, blockSize, 2 *␣
↪blockSize);
    MinMaxBlockComputation c3 = new MinMaxBlockComputation(values, 2 * blockSize, 3 *␣
↪blockSize);
    MinMaxBlockComputation c4 = new MinMaxBlockComputation(values, 3 * blockSize, values.
↪length);
    Thread t1 = new Thread(c1);
    Thread t2 = new Thread(c2);
```

(continues on next page)

(continued from previous page)

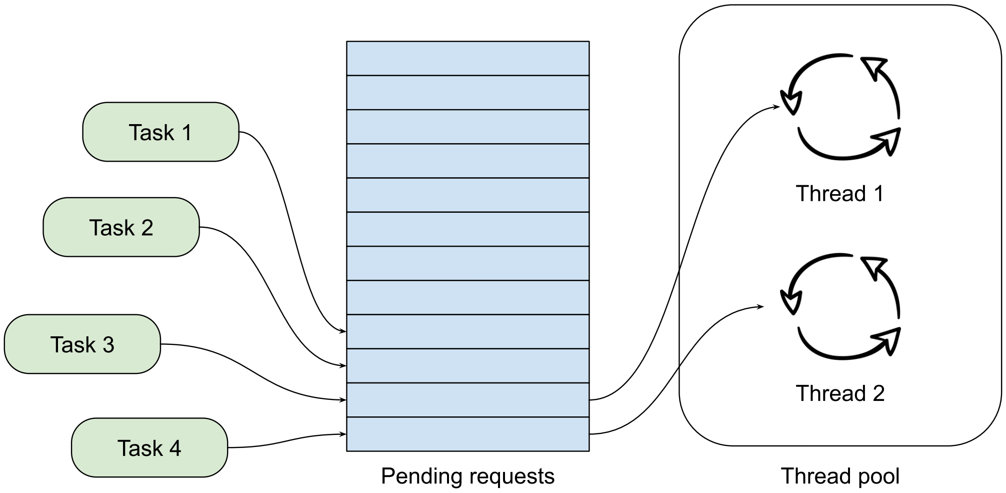
```
    Thread t3 = new Thread(c3);
    Thread t4 = new Thread(c4);
    t1.start();
    t2.start();
    t3.start();
    t4.start();
    t1.join();
    t2.join();
    t3.join();
    t4.join();

    MinMaxResult result = MinMaxResult.empty();
    result.combine(c1.getResult());
    result.combine(c2.getResult());
    result.combine(c3.getResult());
    result.combine(c4.getResult());
    result.print();
}
```

Note that the definition of `c4` uses the size of the array (i.e., `values.length`) as its stop index, instead of `4 * blockSize`, in order to be sure that the last items in the array get processed if the size of the array is not a multiple of 4.

We could continue adding more threads in this way (for instance, 8, 16, 32…). But if we use, say, 100 threads, does that mean that our program will run 100 faster? The answer is no, for at least two reasons:

- Obviously, the level of parallelism is limited by the number of CPU cores that are available. If using a CPU with 4 cores, you cannot expect a speed up of more than 4.
- Even if threads are lightweight, there is still an overhead associated with the creation and management of a thread. On a modern computer, creating a simple thread (without any extra object) takes around 0.05-0.1 ms. That is approximately the time to calculate the sum from 1 to 100,000.

We can conclude that threads only improve the speed of a program if the tasks for the threads are longer than the overhead to create and manage them. This motivates the introduction of **thread pools**. A thread pool is a group of threads that are ready to work:



In this drawing, we have a thread pool that is made of 2 threads. Those threads are continuously monitoring a queue of pending tasks. As soon as some task is enqueued and as soon as some thread becomes available, the available thread takes care of this task. Once the task is over, the thread informs the caller that the result of the task is available, then it goes back to listening to the queue, waiting for a new task to be processed.

Thread pools are an efficient way to avoid the overhead associated with the initialization and finalization of threads. It also allows to write user code that is uncoupled from the number of threads or from the number of CPU cores.

5.4.1 Thread pools in Java

In Java, three different interfaces are generally combined to create a thread pool:

- `java.util.concurrent.ExecutorService` implements the thread pool itself, including the queue of requests and its background threads: https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ExecutorService.html.
- `java.util.concurrent.Callable<T>` is a generic interface that represents the task to be run. The task must return an object of type `T`: https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/Callable.html.
- `java.util.concurrent.Future<T>` is a generic interface that represents the result of a task that is in the process of being computed: https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/Future.html.

The *Java Development Kit (JDK)* provides concrete implementations of `ExecutorService` and `Future`, so we (fortunately!) do not have to implement them by ourselves. A concrete thread pool can be created as follows:

```
ExecutorService executor = Executors.newFixedThreadPool(4 /* numberOfThreads */);
```

As developers, our sole responsibility consists in choosing the generic type `T` and in providing an implementation of interface `Callable<T>` that describes the task to be achieved. The interface `Callable<T>` looks as follows:

```
public interface Callable<T> {
    public T call();
}
```

This looks extremely similar to the `Runnable` interface that *we have been using so far*! The difference between the `Runnable` and a `Callable<T>` interfaces is that the former has no return value, whereas the latter returns a result of type `T`.

Once a concrete implementation of `Callable<T>` is available, tasks can be submitted to the thread pool. The pattern is as follows:

```
Future<T> future1 = executor.submit(new MyCallable(...));
```

Threads in thread pool are like chefs in the kitchen of a restaurant waiting for orders. If you submit one task to the pool using the call above, one of the chefs will take the task and it will immediately start working on it. You can submit more tasks, but they might have to wait until one chef has finished dealing with its current task:

```
Future<T> future2 = executor.submit(new MyCallable(...));
Future<T> future3 = executor.submit(new MyCallable(...));
Future<T> future4 = executor.submit(new MyCallable(...));
// ...
```

You can obtain the result of the futures with their `get()` method:

```
T result1 = future1.get();
T result2 = future2.get();
T result3 = future3.get();
T result4 = future4.get();
// ...
```

If the task is not finished yet, the method `get()` will wait. This contrast with the `executor.submit()` method that always returns immediately.

At the end of the program or when you do not need the thread pool anymore, you have to shut it down explicitly to stop all its threads, otherwise the software might not properly exit:

```
executor.shutdown();
```

5.4.2 Thread pool for computing the minimum and maximum

It is straightforward to turn the `MinMaxBlockComputation` runnable that *was defined above* into an callable:

```
class MinMaxBlockCallable implements Callable<MinMaxResult> {
    private float[] values;
    private int startIndex;
    private int endIndex;
    // Removed member: MinMaxResult result;

    public MinMaxBlockCallable(float[] values,
                               int startIndex,
                               int endIndex) {
        this.values = values;
        this.startIndex = startIndex;
        this.endIndex = endIndex;
    }

    @Override
    public MinMaxResult call() {
        if (startIndex >= endIndex) {
            return MinMaxResult.empty();
        } else {
            float minValue = values[startIndex];
            float maxValue = values[startIndex];

            for (int i = startIndex + 1; i < endIndex; i++) {
                minValue = Math.min(minValue, values[i]);
                maxValue = Math.max(maxValue, values[i]);
            }

            return new MinMaxResult(minValue, maxValue);
        }
    }
}
```

The only differences are:

- The `Runnable` interface is replaced by the `Callable<MinMaxResult>` interface.
- The method `run()` is replaced by method `call()`.
- The member variable `result` and the method `getResult()` are removed. These elements are replaced by the return value of `call()`.

Thanks to the newly defined `MinMaxBlockCallable` class, it is now possible to use a thread pool:

```java
public static void main(String[] args) throws InterruptedException, ExecutionException {
    // Create a thread pool with 4 threads (the thread pool could be shared with other methods)
    ExecutorService executor = Executors.newFixedThreadPool(4);

    float values[] = new float[1024];
    // Fill the array

    // Create two tasks that work on two distinct parts of the whole array
    Future<MinMaxResult> partialResult1 = executor.submit(new MinMaxBlockCallable(values, 0, values.length / 2));
    Future<MinMaxResult> partialResult2 = executor.submit(new MinMaxBlockCallable(values, values.length / 2, values.length));

    // Combine the partial results on the two parts to get the final result
    MinMaxResult finalResult = MinMaxResult.empty();
    finalResult.combine(partialResult1.get());  // This call blocks the main thread until the first part is processed
    finalResult.combine(partialResult2.get());  // This call blocks the main thread until the second part is processed
    finalResult.print();

    // Do not forget to shut down the thread pool
    executor.shutdown();
}
```

This solution looks extremely similar to the previous solution using Runnable and Thread. However, in this code, we do not have to manage the threads by ourselves, and the thread pool could be shared with other parts of the software.

The throws construction is needed because the get() method of futures can possibly throw an InterruptedException (if the future was interrupted while waiting) or an ExecutionException (if there was a problem during the calculation).

5.4.3 Dividing the array into multiple blocks

So far, we have divided the array values into 2 or 4 blocks, because we were guided by the number of CPU cores. In practice, it is a better idea to divide the array into blocks of a fixed size to become agnostic of the underlying number of cores. A thread pool can be used in this situation to manage the computations, while preventing the number of threads to exceed the CPU capacity.

To this end, we can create a separate data structure (e.g., a stack or a list) that keeps track of the pending computations by storing the Future<MinMaxResult> objects:

```java
public static void main(String[] args) throws InterruptedException, ExecutionException {
    ExecutorService executor = Executors.newFixedThreadPool(4);

    float values[] = new float[1024];
    // Fill the array

    int blockSize = 128;

    Stack<Future<MinMaxResult>> pendingComputations = new Stack<>();

```

(continues on next page)

(continued from previous page)

```
    for (int block = 0; block < numberOfBlocks; block++) {
        int startIndex = block * blockSize;
        int endIndex;
        if (block == numberOfBlocks - 1) {
            endIndex = values.length;
        } else {
            endIndex = (block + 1) * blockSize;
        }

        pendingComputations.add(executor.submit(new MinMaxBlockCallable(values,␣
↪startIndex, endIndex)));
    }

    MinMaxResult result = MinMaxResult.empty();

    while (!pendingComputations.empty()) {
        Future<MinMaxResult> partialResult = pendingComputations.pop();
        result.combine(partialResult.get());
    }

    result.print();

    executor.shutdown();
}
```

Note that the end index of the last block is treated specifically, because `values.length` might not be an integer multiple of `blockSize`.

5.4.4 Computing the mean of an array

Up to now, this chapter has been almost entirely focused on the task of finding the minimum and maximum values in an array. We have explained how the introduction of the separate class `MinMaxResult` that is dedicated to the storage of partial results leads to a natural use of thread pools by implementing the `Callable<MinMaxResult>` interface. An important trick was to define the `combine()` method that is responsible for combining the partial results obtained from different parts of the array.

How could we compute the mean of the array using a similar approach?

The first thing is to define a class that stores the partial result over a block of the array. One could decide to store only the mean value itself. Unfortunately, this choice would not give enough information to implement the `combine()` method. Indeed, in order to combine two means, it is necessary to know the number of elements upon which the individual means were computed.

The solution consists in storing the sum and the number of elements in a dedicated class:

```
class MeanResult {
    private double sum;  // We use doubles as we might be summing a large number of␣
↪floats
    private int count;

    public MeanResult() {
        sum = 0;
        count = 0;
```

(continues on next page)

(continued from previous page)

```java
    }

    public void addValue(float value) {
        sum += value;
        count++;
    }

    public boolean isPresent() {
        return count > 0;
    }

    public float getMean() {
        if (isPresent()) {
            return (float) (sum / (double) count);
        } else {
            throw new IllegalStateException();
        }
    }

    public void combine(MeanResult with) {
        sum += with.sum;
        count += with.count;
    }

    public void print() {
        if (isPresent()) {
            System.out.println(getMean());
        } else {
            System.out.println("Empty array");
        }
    }
}
```

Thanks to the `MeanResult` class, the source code of `MinMaxBlockCallable` can be adapted in order to define a callable that computes the mean of a block of an array:

```java
class MeanUsingCallable implements Callable<MeanResult> {
    private float[] values;
    private int startIndex;
    private int endIndex;

    public MeanUsingCallable(float[] values,
                             int startIndex,
                             int endIndex) {
        this.values = values;
        this.startIndex = startIndex;
        this.endIndex = endIndex;
    }

    @Override
    public MeanResult call() {
        MeanResult result = new MeanResult();
```

(continues on next page)

(continued from previous page)

```
        for (int i = startIndex; i < endIndex; i++) {
            result.addValue(values[i]);
        }
        return result;
    }
}
```

This callable can be used as a drop-in replacement in the *source code to compute the minimum/maximum*.

Exercise

The classes MinMaxBlockCallable and MeanUsingCallable share many similarities: They both represent a computation that can be done on a part of an array, they both use a dedicated class to store their results, and they both support the operation combine() to merge partial results. However, the *source code to compute the minimum/maximum* must be adapted for each of them.

Implement a hierarchy of classes/interfaces that can be used to implement a single source code that is compatible with both MinMaxBlockCallable and MeanUsingCallable. Furthermore, validate your approach by demonstrating its compatibility with the computation of the standard deviation.

Hint: Standard deviation can be derived from the variance, which can be computed from the number of elements in the block, from the sum of elements in the block, and from the sum of the squared elements in the block: https://en.wikipedia.org/wiki/Algorithms_for_calculating_variance (cf. naive algorithm).

5.5 Shared memory

In the solutions presented so far, the strategy was to make Runnable or Callable<T> responsible for computing the partial results, then to make the main Java thread responsible to combine those partial results. But, *as explained earlier*, threads that belong to the same process share the same memory space. This means that **threads can access the same variables**.

5.5.1 Using a shared variable to collect the partial results

According to this discussion, it should be possible to make the threads merge *directly* their partial results into a shared variable, freeing the main thread from this combination task. This is a perfectly valid idea, that is implemented in the following Runnable:

```
class SharedMinMaxComputation implements Runnable {
    private SharedMinMaxResult target;
    private float[] values;
    private int startIndex;
    private int endIndex;

    public SharedMinMaxComputation(SharedMinMaxResult target,
                                   float[] values,
                                   int startIndex,
                                   int endIndex) {
        this.target = target;
        this.values = values;
        this.startIndex = startIndex;
```

(continues on next page)

(continued from previous page)

```
        this.endIndex = endIndex;
    }

    @Override
    public void run() {
        if (startIndex < endIndex) {
            float minValue = values[startIndex];
            float maxValue = values[startIndex];

            for (int i = startIndex + 1; i < endIndex; i++) {
                minValue = Math.min(minValue, values[i]);
                maxValue = Math.max(maxValue, values[i]);
            }

            target.combine(new SharedMinMaxResult(minValue, maxValue));
        }
    }
}
```

The `SharedMinMaxComputation` can be executed exactly the same way as `MinMaxBlockComputation` (cf. *above*), except that the main Java thread has to create the `target` variable and to provide it to the individual `Runnable`:

```
public static void main(String[] args) throws InterruptedException {
    float values[] = new float[1024];
    // Fill the array

    SharedMinMaxResult result = SharedMinMaxResult.empty();

    SharedMinMaxComputation c1 = new SharedMinMaxComputation(result, values, 0, values.
↪length / 2);
    SharedMinMaxComputation c2 = new SharedMinMaxComputation(result, values, values.
↪length / 2, values.length);
    Thread t1 = new Thread(c1);
    Thread t2 = new Thread(c2);
    t1.start();
    t2.start();
    t1.join();
    t2.join();

    // No more call to "combine()" here!
    result.print();
}
```

Note that it does not make much sense to use a `Callable<T>` in this approach. Indeed, because the individual computations directly merge their partial results with a shared variable, they never have to report a result to their caller. However, it still makes much sense to use a thread pool to avoid manipulating the threads directly.

This is why the standard interface `ExecutorService` that *implements thread pools* accepts not only implementations of the `Callable<T>` interface in its `submit()` method, but also implementations of the `Runnable` interface. In this case, the futures do not convey any result, so `Future<T>` must simply be replaced by `Future`. This is illustrated in the following code:

```
public static void main(String[] args) throws InterruptedException, ExecutionException {
    float values[] = new float[1024];
    // Fill the array

    ExecutorService executor = Executors.newFixedThreadPool(4);

    SharedMinMaxResult result = SharedMinMaxResult.empty();

    SharedMinMaxComputation c1 = new SharedMinMaxComputation(result, values, 0, values.
↪length / 2);
    SharedMinMaxComputation c2 = new SharedMinMaxComputation(result, values, values.
↪length / 2, values.length);
    Future future1 = executor.submit(c1);
    Future future2 = executor.submit(c2);

    future1.get();  // The "Future" does not convey any result, so we just wait here
    future2.get();

    executor.shutdown();
}
```

5.5.2 Race conditions, mutual exclusion, and monitors

There is however an important danger in the use of shared variables! Let us consider the following code, in which two threads are incrementing the same counter:

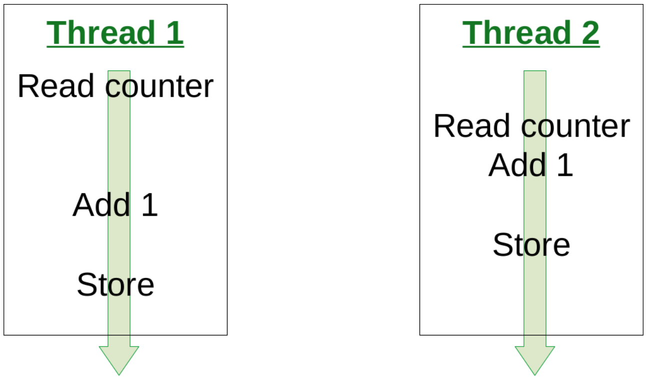
```
public class BadCounter {
    private static int counter = 0; // Both threads use the same counter

    private static void incrementCounter() {
        counter++;
    }

    private static class Counter implements Runnable {
        @Override
        public void run() {
            for (int i = 0; i < 100000; i++) {
                incrementCounter();
            }
        }
    }

    public static void main(String[] args) throws InterruptedException {
        Thread t1 = new Thread(new Counter());
        Thread t2 = new Thread(new Counter());
        t1.start();
        t2.start();
        t1.join();
        t2.join();
        System.out.println(counter);
    }
}
```

One would expect that at the end of the computation, the software would print 200000: The two threads count from 0 to 100000, so the result should be 2 * 100000. However, if you run this software, the printed result is different between each execution and is never equal to 200000 (it is even closer to 100000 than to 200000). Why so?

This is because of **race conditions**. The line counter++ actually corresponds to 3 low-level instructions:

1. Read the value of the variable counter,
2. Add 1 to the read value,
3. Store the incremented value to the variable counter.

This decomposition implies that the following sequence of low-level instructions can happen:



In such a sequence, the first thread would overwrite the change made by the second thread to counter: There is an interference between the two threads! Race conditions depend on the way the instructions are dispatched and ordered between the different CPU cores.

Fortunately, operating systems and thread libraries offer primitives to prevent such race conditions to occur. The idea is to define so-called **critical sections** in the source code, in which at most one thread can be present at any time. In our example, method incrementCounter() should correspond to a critical section: Thread 1 should have waited for thread 2 to write its result to the shared variable before starting its computation.

In Java, critical sections can be defined by adding the synchronized keyword to the methods associated with a shared object. Our example can be made correct simply by replacing:

```java
public void incrementCounter() {
    counter++;
}
```

by:

```java
public synchronized void incrementCounter() {
    counter++;
}
```

Intuitively, adding the synchronized keyword means that a thread entering the method "locks a padlock". As a consequence, any other thread arriving later on cannot enter the method, because the padlock is locked. Once the first thread exits the method, it "unlocks the padlock", leaving the opportunity to one of the waiting threads to enter the method in its turn.

Internally, each Java object is automatically equipped with one padlock that is shared between all the methods of the object. This padlock is referred to as the **monitor** of the object. The process of locking/unlocking the monitor is referred to as **running in mutual exclusion**.

Another reason for using synchronized consists in ensuring the **visibility** of variable modifications done by one thread to the other threads. For instance, let us consider the following source code:

```java
public class Visibility {
    private static boolean ready;
    private static String name = "Hello";

    private static class MyRunnable implements Runnable {
        public void run() {
            while (!ready) {
                // Wait for "ready" to become "true"
            }
            System.out.println(name);
        }
    }

    public static void main(String[] args) {
        Runnable r = new MyRunnable();
        Thread t = new Thread(r);

        t.start();
        name = "World";
        ready = true;
    }
}
```

While it may seem obvious that the software will print `World`, it is in fact possible that it will print `Hello`, or never terminate at all! The problem is that the Java specification does not guarantee that some thread sees the modifications made by another thread, unless both threads synchronize (for example with a `synchronized` statement). In other words, the `synchronized` keyword also implies that threads *publish* their changes to other threads.

In the program above, the `while (!ready)` loop might go on forever because the modification `ready = true` done by the main thread might never become visible to `MyRunnable`. Even worse, the JVM might decide to swap the statements `name = "World"` and `ready = true` in the main thread to apply some low-level hardware optimization: The Java specification allows such *reorderings*, as long as the reordering does not change the semantics of the code within *that* thread. As our source code does not enforce the visibility of the changes to `ready`, the JVM will notice that the reordering has no side-effect on the main thread and might decide to first execute `ready = true`, which will make `MyRunnable` print `Hello` instead of `World`.

The solution is to make sure that the two threads access the shared variables only through `synchronized` methods, as in the following source code:

```java
public class FixedVisibility {
    private static boolean ready;
    private static String name = "Hello";

    public static synchronized void start(String newName) {
        ready = true;
        name = newName;
    }

    public static synchronized boolean isReady() {
        return ready;
    }

    public static synchronized String getName() {
        return name;
```

(continues on next page)

(continued from previous page)

```java
    }

    private static class MyRunnable implements Runnable {
        public void run() {
            while (!isReady()) {
                // Wait for "ready" to become "true"
            }
            System.out.println(getName());
        }
    }

    public static void main(String[] args) {
        Runnable r = new MyRunnable();
        Thread t = new Thread(r);

        t.start();
        start("World");
    }
}
```

Summarizing, as a general rule of thumb in this course, make sure that **shared variables are only accessed through synchronized methods**. This will both prevent race conditions and ensure the visibility of the changes.

There is one main caveat associated with the `synchronized` keyword: Because `synchronized` methods limit the concurrency between threads, they also reduce the overall performance of multithreaded software. In other words, synchronized methods represent a **bottleneck** in the execution of a multithreaded software. For instance, you should try and avoid making a complex computation in a `synchronized` method, if possible. Nevertheless, always remember that it is less important to have a program that is *fast* than a program that *works properly*! **Correctness is always more important than speed**.

Finally, note that the `synchronized` keyword is only the most basic of the multiple synchronization mechanisms for multithreading that are offered by Java. It is however the most important one, as it allows one to develop thread-safe software without diving too much into the complexity of parallel programming. As such, `synchronized` should be mastered by any Java developer. As this course is about the basics of multithreading, more advanced constructions will not be covered.

5.5.3 Shared object for computing the minimum and maximum

We now apply mutual exclusion to our example of *computing the minimum and maximum using a shared variable*. Note that in the version that uses the shared `result` variable, the original class `MinMaxResult` was replaced by the `SharedMinMaxResult` class. The latter class is exactly the same as `MinMaxResult`, but with the addition of the `synchronized` keyword in each of its methods:

```java
class SharedMinMaxResult {
    private boolean isPresent;
    private float minValue;
    private float maxValue;

    private SharedMinMaxResult(boolean isPresent,
                               float minValue,
                               float maxValue) {
        this.isPresent = isPresent;
```

(continues on next page)

(continued from previous page)

```
        this.minValue = minValue;
        this.maxValue = maxValue;
    }

    public SharedMinMaxResult(float minValue,
                              float maxValue) {
        this(true /* present */, minValue, maxValue);
    }

    static public SharedMinMaxResult empty() {
        return new SharedMinMaxResult(false /* not present */, 0 /* dummy min */, 0 /* dummy max */);
    }

    public synchronized boolean isPresent() {
        return isPresent;
    }

    public synchronized float getMinValue() {
        if (isPresent()) {
            return minValue;
        } else {
            throw new IllegalStateException();
        }
    }

    public synchronized float getMaxValue() {
        if (isPresent()) {
            return maxValue;
        } else {
            throw new IllegalStateException();
        }
    }

    public synchronized void print() {
        if (isPresent()) {
            System.out.println(getMinValue() + " " + getMaxValue());
        } else {
            System.out.println("Empty array");
        }
    }

    public synchronized void combine(SharedMinMaxResult with) {
        if (with.isPresent) {
            if (isPresent) {
                // Combine the results from two non-empty blocks
                minValue = Math.min(minValue, with.minValue);
                maxValue = Math.max(maxValue, with.maxValue);
            } else {
                // Replace the currently absent result by the provided result
                isPresent = true;
                minValue = with.minValue;
```

(continues on next page)

(continued from previous page)

```
                maxValue = with.maxValue;
            }
        } else {
            // Do nothing if the other result is absent
        }
    }
}
```

The addition of synchronized prevents any race condition between the threads when they combine their partial results with the final result. Also note that the synchronized methods are only part of the class holding the final result: The threads still make their computation in full parallelism. It is only when the partial results are combined that mutual exclusion occurs, which does not represent a large bottleneck.

The downside of this source code is that the classes MinMaxResult and SharedMinMaxResult share almost all of their source code, which is highly redundant. One could avoid this redundancy by wrapping an object of class MinMaxResult inside a class with the same set of methods, but with the synchronized keyword added:

```
class SharedMinMaxResult {
    private MinMaxResult wrapped;

    private SharedMinMaxResult(MinMaxResult wrapped) {
        this.wrapped = wrapped;
    }

    public SharedMinMaxResult(float minValue,
                               float maxValue) {
        this(new MinMaxResult(minValue, maxValue));
    }

    static public SharedMinMaxResult empty() {
        return new SharedMinMaxResult(MinMaxResult.empty());
    }

    public synchronized boolean isPresent() {
        return wrapped.isPresent();
    }

    public synchronized float getMinValue() {
        return wrapped.getMinValue();
    }

    public synchronized float getMaxValue() {
        return wrapped.getMaxValue();
    }

    public synchronized void print() {
        wrapped.print();
    }

    public synchronized void combine(SharedMinMaxResult with) {
        wrapped.combine(with.wrapped);
    }
}
```

5.5.4 Application to matrix multiplication

Linear algebra is a mathematical domain that can greatly benefit from parallel programming. This section gives an example about how multithreading can be used to implement matrix multiplication.

Let us consider the following basic implementation of a matrix in Java:

```java
public class SynchronizedMatrix {
    private float values[][];

    private void checkPosition(int row,
                               int column) {
        if (row >= getRows() ||
            column >= getColumns()) {
            throw new IllegalArgumentException();
        }
    }

    public SynchronizedMatrix(int rows,
                              int columns) {
        if (rows <= 0 ||
            columns <= 0) {
            throw new IllegalArgumentException();
        } else {
            values = new float[rows][columns];
        }
    }

    public synchronized int getRows() {
        return values.length;
    }

    public synchronized int getColumns() {
        // "values[0]" is guaranteed to exist, because "columns > 0" in the constructor
        return values[0].length;
    }

    public synchronized float getValue(int row,
                                       int column) {
        checkPosition(row, column);
        return values[row][column];
    }

    public synchronized void setValue(int row,
                                      int column,
                                      float value) {
        checkPosition(row, column);
        values[row][column] = value;
    }
}
```

Note that the SynchronizedMatrix class has each of its methods tagged with the synchronized keyword, so it is suitable for use as a shared variable between multiple threads.

We are interested in the task of computing the product $C \in \mathbb{R}^{m \times p}$ of two matrices $A \in \mathbb{R}^{m \times n}$ and $B \in \mathbb{R}^{n \times p}$ of compatible dimensions. Remember the definition of matrix multiplication: $c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$. This definition can directly be turned into a sequential algorithm:

```
public static void main(String args[]) {
    SynchronizedMatrix a = new SynchronizedMatrix(..., ...);
    SynchronizedMatrix b = new SynchronizedMatrix(..., ...);
    // Fill a and b

    SynchronizedMatrix c = new SynchronizedMatrix(a.getRows(), b.getColumns());

    for (int row = 0; row < c.getRows(); row++) {
        for (int column = 0; column < c.getColumns(); column++) {
            float accumulator = 0;
            for (int k = 0; k < a.getColumns(); k++) {
                accumulator += a.getValue(row, k) * b.getValue(k, column);
            }
            c.setValue(row, column, accumulator);
        }
    }
}
```

How could we leverage multithreading to speed up this computation? The main observation is that the innermost loop on k is executed for each entry of C. Therefore, a possible solution consists in creating $m \times p$ tasks that will be processed by a thread pool, each of these tasks implementing the innermost loop with the accumulator.

A first possible implementation consists in creating a "result" data structure that will store the value of one cell of the matrix product:

```
class ProductAtCellResult {
    private int row;
    private int column;
    private float value;

    ProductAtCellResult(int row,
                        int column,
                        float value) {
        this.row = row;
        this.column = column;
        this.value = value;
    }

    public int getRow() {
        return row;
    }

    public int getColumn() {
        return column;
    }

    public float getValue() {
        return value;
    }
}
```

We can then implement the `Callable<T>` interface to use this data structure in a thread pool:

```
static class ComputeProductAtCell implements Callable<ProductAtCellResult> {
    private SynchronizedMatrix a;
    private SynchronizedMatrix b;
    private int row;
    private int column;

    public ComputeProductAtCell(SynchronizedMatrix a,
                                SynchronizedMatrix b,
                                int row,
                                int column) {
        this.a = a;
        this.b = b;
        this.row = row;
        this.column = column;
    }

    public ProductAtCellResult call() {
        float accumulator = 0;

        for (int k = 0; k < a.getColumns(); k++) {
            accumulator += a.getValue(row, k) * b.getValue(k, column);
        }

        return new ProductAtCellResult(row, column, accumulator);
    }
}
```

The thread pool would then be used as follows:

```
public static void main(String args[]) throws ExecutionException, InterruptedException {
    SynchronizedMatrix a = new SynchronizedMatrix(..., ...);
    SynchronizedMatrix b = new SynchronizedMatrix(..., ...);
    // Fill a and b

    ExecutorService executor = Executors.newFixedThreadPool(10);

    Stack<Future<ProductAtCellResult>> pendingComputations = new Stack<>();

    for (int row = 0; row < a.getRows(); row++) {
        for (int column = 0; column < b.getColumns(); column++) {
            pendingComputations.add(executor.submit(new ComputeProductAtCell(a, b, row,␣
↪column)));
        }
    }

    SynchronizedMatrix c = new SynchronizedMatrix(a.getRows(), b.getColumns());

    while (!pendingComputations.empty()) {
        Future<ProductAtCellResult> future = pendingComputations.pop();
        c.setValue(future.get().getRow(), future.get().getColumn(), future.get().
↪getValue());
    }

```

(continues on next page)

(continued from previous page)

```
    executor.shutdown();
}
```

Note that this implementation would also work fine if there were no `synchronized` keyword in the class implementing matrices, because the threads do not apply modifications to shared variables. Even though this solution works fine, it is demanding in terms of memory. Indeed, the row, column, and value of each cell in the product will first be stored in a separate data structure (the stack of futures referring to `ProductAtCellResult` objects), before being copied onto the target matrix `c`. Couldn't the results be directly written into `c`?

The answer is obviously "yes": Thanks to the fact that the `SynchronizedMatrix` implements the `synchronized` keyword, it is thread-safe and it can be used as a shared variable into which the different threads can directly write their results.

According to this discussion, here is a second possible implementation that replaces `Callable<ProductAtCellResult>` by a set of `Runnable` objects, each of those objects writing one cell of the target matrix:

```java
static class StoreProductAtCell implements Runnable {
    private SynchronizedMatrix a;
    private SynchronizedMatrix b;
    private SynchronizedMatrix c;
    private int row;
    private int column;

    public StoreProductAtCell(SynchronizedMatrix a,
                              SynchronizedMatrix b,
                              SynchronizedMatrix c,
                              int row,
                              int column) {
        this.a = a;
        this.b = b;
        this.c = c;
        this.row = row;
        this.column = column;
    }

    public void run() {
        float accumulator = 0;

        for (int k = 0; k < a.getColumns(); k++) {
            accumulator += a.getValue(row, k) * b.getValue(k, column);
        }

        c.setValue(row, column, accumulator);
    }
}
```

Thanks to the `StoreProductAtCell` class, the matrix multiplication can be computed by a thread pool as follows:

```java
public static void main(String args[]) throws ExecutionException, InterruptedException {
    SynchronizedMatrix a = new SynchronizedMatrix(..., ...);
    SynchronizedMatrix b = new SynchronizedMatrix(..., ...);
    // Fill a and b
```

(continues on next page)

(continued from previous page)

```
    ExecutorService executor = Executors.newFixedThreadPool(10);

    Stack<Future> pendingComputations = new Stack<>();

    SynchronizedMatrix c = new SynchronizedMatrix(a.getRows(), b.getColumns());

    for (int row = 0; row < a.getRows(); row++) {
        for (int column = 0; column < b.getColumns(); column++) {
            pendingComputations.add(executor.submit(new StoreProductAtCell(a, b, c, row,␣
↪column)));
        }
    }

    while (!pendingComputations.empty()) {
        Future future = pendingComputations.pop();
        future.get();
    }

    executor.shutdown();
}
```

Note that in the specific example of matrix multiplication, the solution based on `Runnable` is both simpler and more memory efficient, as it does not require the use of an intermediate class such as `ProductAtCellResult` to store the partial results before writing them onto the target matrix.

Remark

In practice, the parallel implementation of matrix multiplication proposed above will probably have really bad performance, possibly even worse than a purely sequential algorithm. Indeed, this implementation totally neglects the processor cache: Because the various threads work on different areas of the matrices, the CPU will continuously have to access the RAM, which slows down the computation. Real software libraries for optimized linear algebra implement algorithms that feature **locality** in their patterns of access to the RAM. Because CPU cores include a cache memory that is much faster than the RAM, exploiting locality can dramatically increase the performance of an algorithm. Such optimized algorithms will typically leverage block matrix multiplication and SIMD instructions, in addition to multithreading.

CHAPTER

SIX

FUNCTIONAL PROGRAMMING

Functional programming refers to a programming paradigm that emphasizes the **use of functions and immutable data** to create applications. This paradigm promotes writing code that is easier to reason about, and that allows for better handling of concurrency.

While Java is not a pure functional language like Haskell, it offers many features that can be used to write more functional-style code. Functional programming in Java encourages the use of pure functions that have no side effect, i.e., that avoid changing the state of the program. Java 8 introduced features to support functional programming, primarily through the addition of functional interfaces, of lambda expressions, and of the `Stream` API.

6.1 Nesting classes

To begin our study of functional programming, we will first go back to the concept of nested classes that has previously been *briefly encountered*. A nested class is simply a **class that is defined within another class**. Note that a nested class can also define its own nested classes, just like Matryoshka dolls.

Let us consider the task of creating a spreadsheet application. A spreadsheet document is composed of a number of rows. Each row is made of several columns with string values. A data structure to represent a single row can be modeled as follows:

```java
import java.util.HashMap;
import java.util.Map;

class Row {
    private Map<Integer, String> columns = new HashMap<>();

    public void put(int column,
                    String value) {
        columns.put(column, value);
    }

    public String get(int column) {
        return columns.getOrDefault(column, "" /* default value if absent */);
    }
}
```

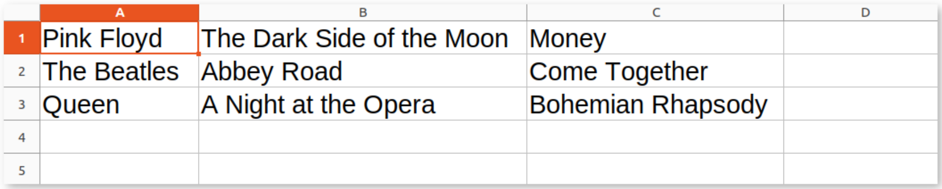
The `Row` class uses an associative array that maps integers (the index of the columns) to strings (the value of the columns). The use of an associated array allows to account for columns with a missing value. The standard `HashMap<K, V>` class is used to this end: https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html

A basic spreadsheet application can then be created on the top of this `Row` class. Let us define a spreadsheet document as an ordered list of rows:

```java
import java.util.ArrayList;
import java.util.List;

public class Spreadsheet {
    private List<Row> rows;
    private int sortOnColumn;

    public Spreadsheet() {
        this.rows = new ArrayList<>();
        this.sortOnColumn = 0;
    }

    public void add(Row row) {
        rows.add(row);
        sort();
    }

    public void setSortOnColumn(int sortOnColumn) {
        this.sortOnColumn = sortOnColumn;
        sort();
    }

    private void sort() {
        // We will implement this later on
    }

    static private void fillWithSongs(Spreadsheet spreadsheet) {
        Row row = new Row();
        row.put(0, "Pink Floyd");
        row.put(1, "The Dark Side of the Moon");
        row.put(2, "Money");
        spreadsheet.add(row);

        row = new Row();
        row.put(0, "The Beatles");
        row.put(1, "Abbey Road");
        row.put(2, "Come Together");
        spreadsheet.add(row);

        row = new Row();
        row.put(0, "Queen");
        row.put(1, "A Night at the Opera");
        row.put(2, "Bohemian Rhapsody");
        spreadsheet.add(row);
    }

    static public void main(String[] args) {
        Spreadsheet spreadsheet = new Spreadsheet();
        fillWithSongs(spreadsheet);
    }
}
```

This Java application creates a spreadsheet with 3 rows and 3 columns that are filled with information about 3 songs.

If exported to a real-world spreadsheet application such as LibreOffice Calc, it would be rendered as follows:



	A	B	C	D
1	Pink Floyd	The Dark Side of the Moon	Money	
2	The Beatles	Abbey Road	Come Together	
3	Queen	A Night at the Opera	Bohemian Rhapsody	
4				
5				

6.1.1 Static nested classes

We are now interested in the task of continuously sorting the rows according to the values that are present in the columns, as new rows get added to the spreadsheet using the addRow() method.

To this end, the Spreadsheet class contains the member variable sortOnColumn that specifies on which column the sorting must be applied. That parameter can be set using the setSortOnColumn() setter method. We already know that the task of sorting the rows can be solved through *delegation to a dedicated comparator*:

```java
class RowComparator1 implements Comparator<Row> {
    private int column;

    RowComparator1(int column) {
        this.column = column;
    }

    @Override
    public int compare(Row a, Row b) {
        return a.get(column).compareTo(b.get(column));
    }
}

public class Spreadsheet {
    private List<Row> rows;
    private int sortOnColumn;
    // ...

    private void sort() {
        Collections.sort(rows, new RowComparator1(sortOnColumn));
    }
}
```

The RowComparator1 class is called an **external class** because it is located outside of the Spreadsheet class. This is not an issue because this sample code is quite short. But in real code, it might be important for readability to bring the comparator class closer to the method that uses it (in this case, sort()). This is why Java features **static nested classes**. This construction allows to define a class at the member level of another class:

```java
public class Spreadsheet {
    private List<Row> rows;
    private int sortOnColumn;
    // ...

    private static class RowComparator2 implements Comparator<Row> {
        private int column;
```

(continues on next page)

(continued from previous page)

```
        RowComparator2(int column) {
            this.column = column;
        }

        @Override
        public int compare(Row a, Row b) {
            return a.get(column).compareTo(b.get(column));
        }
    }

    private void sort() {
        Collections.sort(rows, new RowComparator2(sortOnColumn));
    }
}
```

In this code, RowComparator2 is the static nested class, and Spreadsheet is called its **outer class**. Note that RowComparator2 could have been tagged with a *public visibility* to make it accessible outside of Spreadsheet, in the case the developer felt like sorting collections of Row objects could make sense in other parts of the application.

Static nested classes are a way to logically group classes together, to improve code organization, and to encapsulate functionality within a larger class. This promotes a more modular and structured design, in a way that is similar to *packages*, but at a finer granularity. Note that it is allowed for two different classes to use the same name for a nested class, which can prevent collisions between class names in large applications.

Importantly, static nested classes have access to the private static members of the outer class, which was not the case of the external class RowComparator1: This can for instance be useful to take advantage of *private enumerations or constants* that would be defined inside the outer class.

6.1.2 Inner classes

The previous code has however a redundancy: The value of sortOnColumn must be manually copied to a private column variable of RowComparator2 so that it can be used inside the compare() method. Can we do better? The answer is “yes”, thanks to the concept of non-static nested classes, that are formally known as **inner classes**. Java allows writing:

```
public class Spreadsheet {
    private List<Row> rows;
    private int sortOnColumn;
    // ...

    private class RowComparator3 implements Comparator<Row> {
        @Override
        public int compare(Row a, Row b) {
            return a.get(column).compareTo(b.get(sortOnColumn));
        }
    }

    private void sort() {
        Collections.sort(rows, new RowComparator3());
    }
}
```

This is much more compact! In this code, `private static class` was simply replaced by `private class`. Thanks to this modification, `RowComparator3` becomes an inner class of the outer class `Spreadsheet`, which grants its `compare()` method a direct access to the `sortOnColumn` member variable.

Inner classes look very similar to static nested classes, but they do not have the `static` keyword. As can be seen, the methods of inner classes can not only access the static member variables of the outer class, but they can also transparently access any member of the object that constructed them (variables and methods, including private members). Note that inner classes were previously encountered in this course when the *implementation of custom iterators* was discussed.

It is tempting to systematically use inner classes instead of static nested classes. But pay attention to the fact that inner classes induce a much closer coupling with their outer classes, which can make it difficult to refactor the application, and which can quickly lead to the so-called Feature Envy "code smell" (i.e. the opposite of a good design pattern). Use an inner class only when you need access to the instance members of the outer class. Use a static nested class when there is no need for direct access to the outer class instance or when you want clearer namespacing and better code organization.

6.1.3 Syntactic sugar

The fact that `compare()` has access to `sortOnColumn` might seem magic. This is actually an example of **syntactic sugar**. Syntactic sugar refers to language features or constructs that do not introduce new functionality but provide a more convenient or expressive way of writing code. These features make the code more readable or more concise without fundamentally changing how it operates. In essence, syntactic sugar is a shorthand or a more user-friendly syntax for expressing something that could be written in a longer or more explicit manner.

Syntactic sugar constructions were already encountered in this course. *Autoboxing* is such a syntactic sugar. Indeed, the code:

```
Integer num = 42;  // Autoboxing (from primitive type to wrapper)
int value = num;   // Auto unboxing (from wrapper to primitive type)
```

is semantically equivalent to the more explicit code:

```
Integer num = Integer.valueOf(42);
int value = num.intValue();
```

Thanks to its knowledge about the internals of the standard `java.lang.Integer` class, the compiler can automatically "fill the dots" by adding the constructor and selecting the proper conversion method. The *enhanced for-each loop for iterators* is another example of syntactic sugar, because writing:

```
List<Integer> a = new ArrayList<>();
a.add(-1);
a.add(10);
a.add(42);

for (Integer item: a) {
    System.out.println(item);
}
```

is semantically equivalent to:

```
Iterator<Integer> it = a.iterator();

while (it.hasNext()) {
    Integer item = it.next();
```

(continues on next page)

(continued from previous page)

```
    System.out.println(item);
}
```

Once the compiler comes across some `for()` loop on a collection that implements the standard `Iterable<E>` interface, it can transparently instantiate the iterator and traverse the collection using this iterator.

In the context of inner classes, the syntactic sugar consists in including a reference to the outer object that created the instance of the inner object. In our example, the compiler automatically transforms the `RowComparator3` class into the following static nested class:

```
public class Spreadsheet {
    private List<Row> rows;
    private int sortOnColumn;
    // ...

    private static class RowComparator4 implements Comparator<Row> {
        private Spreadsheet outer;  // Reference to the outer object

        RowComparator4(Spreadsheet outer) {
            this.outer = outer;
        }

        @Override
        public int compare(Row a, Row b) {
            return a.get(outer.sortOnColumn).compareTo(b.get(outer.sortOnColumn));
        }
    }

    private void sort() {
        Collections.sort(rows, new RowComparator4(this));
    }
}
```

As can be seen, the compiler transparently adds a new argument to the constructor of the inner class, which contains the reference to the outer object.

6.1.4 Local inner classes

So far, we have seen three different constructions to define classes:

- External classes are the default way of defining classes, i.e., separately from any other class.
- Static nested classes are members of an outer class. They have access to the static members of the outer class.
- Inner classes are non-static members of an outer class. They are connected to the object that created them through syntactic sugar.

Inner classes are great for the spreadsheet application, but code readability could still be improved if the `RowComparator3` class could somehow be brought *inside* the `sort()` method, because it is presumably the only location where this comparator would make sense in the application. This would make the one-to-one relation between the method and its comparator immediately apparent. This is the objective of **local inner classes**:

```
private void sort() {
    class RowComparator5 implements Comparator<Row> {
```

(continues on next page)

(continued from previous page)

```java
        @Override
        public int compare(Row a, Row b) {
            return a.get(sortOnColumn).compareTo(b.get(sortOnColumn));
        }
    }

    Collections.sort(rows, new RowComparator5());
}
```

In this new version of the sort() method, the comparator was defined within the scope of the method. The RowComparator5 class is entirely local to sort(), and cannot be used in another method or class, which further reduces coupling.

6.1.5 Anonymous inner classes

Because local inner classes are typically used at one single point of the method, it is generally not useful to give a name to local inner classes (in the previous example, this name was RowComparator5). Consequently, Java features the **anonymous inner class** construction:

```java
private void sort() {
    Comparator<Row> comparator = new Comparator<Row>() {
        @Override
        public int compare(Row a, Row b) {
            return a.get(sortOnColumn).compareTo(b.get(sortOnColumn));
        }
    };

    Collections.sort(rows, comparator);
}
```

As can be seen in this example, an anonymous inner class is a class that is defined without a name inside a method and that instantiated at the same place where it is defined.

This construction is often used for implementing interfaces or extending classes on-the-fly. To make this more apparent, note that we could have avoided the introduction of temporary variable comparator by directly writing:

```java
private void sort() {
    Collections.sort(rows, new Comparator<Row>() {
        @Override
        public int compare(Row a, Row b) {
            return a.get(sortOnColumn).compareTo(b.get(sortOnColumn));
        }
    });
}
```

Anonymous inner classes also correspond to another *syntactic sugar* construction, because an anonymous inner class can easily be converted into a local inner class by giving it a meaningless name.

6.1.6 Access to method variables

Importantly, both local inner classes and anonymous inner classes have **access to the variables of their enclosing method**.

To illustrate this point, let us consider the task of filling a matrix with a constant value using multiple threads. We could create one thread that fills the upper part of the matrix, and another thread that fills the lower part of the matrix. Using a *thread pool* and the `SynchronizedMatrix` class that was defined to *demonstrate multithreading*, the corresponding implementation could be:

```java
public static void fill1(ExecutorService threadPool,
                         SynchronizedMatrix m,
                         float value) throws ExecutionException, InterruptedException {

    class Filler implements Runnable {
        private int startRow;
        private int endRow;

        Filler(int startRow,
               int endRow) {
            this.startRow = startRow;
            this.endRow = endRow;
        }

        @Override
        public void run() {
            for (int row = startRow; row < endRow; row++) {
                for (int column = 0; column < m.getColumns(); column++) {
                    // The inner class has access to the "m" and "value" variables!
                    m.setValue(row, column, value);
                }
            }
        }
    }

    Future upperPart = threadPool.submit(new Filler(0, m.getRows() / 2));
    Future lowerPart = threadPool.submit(new Filler(m.getRows() / 2, m.getRows()));

    upperPart.get();
    lowerPart.get();
}
```

As can be seen in this example, it is not necessary for the inner class `Filler` to explicitly store a copy of `m` and `value`. Indeed, because those two variables are part of the scope of method `fill1()`, the `run()` method has direct access to the `m` and `value` variables. Actually, this is again *syntactic sugar*: The compiler automatically gives a **copy of all the local variables of the surrounding method** to the constructor of the inner class.

The method `fill1()` creates exactly two threads, one for each part of the matrix. One could want to take advantage of a higher number of CPU cores by reducing this granularity. According to this idea, here is an alternative implementation that introduces parallelism at the level of the individual rows of the matrix:

```java
public static void fill2(ExecutorService threadPool,
                         SynchronizedMatrix m,
                         float value) throws ExecutionException, InterruptedException {
    Stack<Future> pendingRows = new Stack<>();
```

(continues on next page)

(continued from previous page)

```
    for (int row = 0; row < m.getRows(); row++) {
        final int myRow = row;

        pendingRows.add(threadPool.submit(new Runnable() {
            @Override
            public void run() {
                for (int column = 0; column < m.getColumns(); column++) {
                    m.setValue(myRow, column, value);
                }
            }
        }));
    }

    while (!pendingRows.isEmpty()) {
        pendingRows.pop().get();
    }
}
```

Contrarily to `fill1()` that used a *local* inner class, the `fill2()` method uses an *anonymous* inner class, an instance of which is created for each row. This construction was not possible in the first implementation, because it had to separately track exactly two futures using two variables, which needed to share the definition of the inner class between the two separate runnables. However, in the second implementation, thanks to the fact that the multiple futures are tracked in a uniform way using a stack, the definition of the inner class can occur at a single place.

There is however a caveat associated with `fill2()`: One could expect to have access to the `row` variable inside the `run()` method, because `row` is part of the scope of the enclosing method. However, the inner class might continue to exist and be used even after the loop has finished executing and the variable `row` has disappeared. To prevent potential issues arising from changes to variables after the start of the execution of a method, an inner class is actually only allowed to access the **final variables** in the scope of method (or variables that could have been tagged as `final`). Remember that a final variable means that it is *not allowed to change its value later*.

In the `fill2()` example, `m` and `value` could have been explicitly tagged as `final`, because their value does not change in the method. But adding a line like `value = 10;` inside the method would break the compilation, because `value` could not be tagged as `final` anymore, which would prevent the use of `value` inside the runnable. One could argue that the *content* of `m` changes because of the calls to `m.setValue()`, however the *reference* to the object `m` that was originally provided as argument to the method never changes. Finally, the variable `row` cannot be declared as `final`, because its value changes during the loop. Storing a copy of `row` inside the variable `myRow` is a workaround to solve this issue.

Remark

The example of filling a matrix using multithreading is a bit academic, because for such an operation, the bottleneck will be the RAM, not the CPU. As a consequence, adding more CPU threads will probably never improve performance, and might even be detrimental because of the overhead associated with thread management. Furthermore, our class `SynchronizedMatrix` implements mutual exclusion for the access to the individual cells (i.e. the `setValue()` is tagged with the `synchronized` keyword), which will dramatically reduce the performance.

6.2 Functional interfaces and lambda functions

Since the beginning of our *exploration of object-oriented programming*, a recurrent pattern keeps appearing:

- During the *delegation to comparators of objects*:

```java
public class TitleComparator implements Comparator<Book> {
    @Override
    public int compare(Book b1, Book b2) {
        return b1.getTitle().compareTo(b2.getTitle());
    }
}

// ...
Collections.sort(books, new TitleComparator());
```

- Inside the *Observer Design Pattern*:

```java
class ButtonActionListener implements ActionListener {
    @Override
    public void actionPerformed(ActionEvent e) {
        JOptionPane.showMessageDialog(null,"Thank you!");
    }
}

// ...
button.addActionListener(new ButtonActionListener());
```

- For *specifying operations to be done by threads*:

```java
class Computation implements Runnable {
    @Override
    public void run() {
        expensiveComputation();
    }
}

// ...
Thread t = new Thread(new Computation());
t.start();
```

This recurrent pattern corresponds to simple classes that implement **one single abstract method** and that have **no member**.

The presence of a single method stems from the fact that these classes implement a **single functional interface**. In Java, a functional interface is defined as an *interface* that contains only one abstract method. Functional interfaces are also known as Single Abstract Method (SAM) interfaces. Functional interfaces are a key component of functional programming support introduced in Java 8. The interfaces `Comparator<T>`, `ActionListener`, `Runnable`, and `Callable<T>` are all examples of functional interfaces.

Advanced remarks

A functional interface can have multiple `default` methods or `static` methods without violating the rule of having a single abstract method. This course has not covered `default` methods, but it is sufficient to know that a `default` method provides a default implementation within an interface that the classes implementing the interface can choose to inherit or overwrite. For instance, the interface `Comparator<T>` comes with multiple `default` and `static` methods, as can be seen in the Java documentation: https://docs.oracle.com/javase/8/docs/api/java/util/Comparator.html

In Java 8 and later, the `@FunctionalInterface` annotation helps explicitly mark an interface as a functional interface. If an interface annotated with `@FunctionalInterface` contains more than one abstract method, the compiler generates an error to indicate that it does not meet the criteria of a functional interface. Nonetheless, pay attention to the fact that not all the functional interfaces of Java are annotated with `@FunctionalInterface`. This is notably the case of `ActionListener`.

A **lambda expression** is an expression that creates an instance of an *anonymous inner class* that has no member and that implements a functional interface. Thanks to lambda expressions, the `sort()` method for our spreadsheet application can be shortened as a single line of code:

```java
private void sort() {
    Collections.sort(rows, (a, b) -> a.get(sortOnColumn).compareTo(b.get(sortOnColumn)));
}
```

As can be seen in this example, a lambda expression only specifies the name of the arguments and the body of the single abstract method of the functional interface it implements.

A lambda expression can only appear in a context that expects a value whose type is a functional interface. Once the Java compiler has determined which functional interface is expected for this context, it transparently instantiates a suitable anonymous inner class that implements the expected functional interface with the expected single method.

Concretely, in the `sort()` example, the compiler notices the construction `Collections.sort(rows, lambda)`. Because `rows` has type `List<Row>`, the compiler looks for a static method in the `Collections` class that is named `sort()` and that takes as arguments a value of type `List<Row>` and a functional interface. As can be seen in the Java documentation, the only matching method is `Collections.sort(List<T> list, Comparator<? super T> c)`, with `T` corresponding to class `Row`. The compiler deduces that the functional interface of interest is `Comparator<Row>`, and it accordingly creates an anonymous inner class as follows:

```java
private void sort() {
    // "Comparator<Row>" is the functional interface that matches the lambda expression
    Collections.sort(rows, new Comparator<Row>() {
        @Override
        // The name of the single abstract method and the types of the arguments
        // are extracted from the functional interface. The name of the arguments
        // are taken from the lambda expression.
        public int compare(Row a, Row b) {
            // This is the body of the lambda expression
            return a.get(sortOnColumn).compareTo(b.get(sortOnColumn));
        }
    }
}
```

In other words, lambda expressions are also *syntactic sugar*! Very importantly, **functional interfaces provide a clear contract for the signature of the method that the matching lambda expression must implement**, which is needed for this syntactic sugar to work.

Thanks to lambda expressions, the three examples at the beginning of this section could all be simplified as one-liners:

- During the *delegation to comparators of objects*:

  ```java
  Collections.sort(books, (b1, b2) -> b1.getTitle().compareTo(b2.getTitle()));
  ```

- Inside the *Observer Design Pattern*:

```
button.addActionListener(() -> JOptionPane.showMessageDialog(null,"Thank you!"));
```

- For *specifying operations to be done by threads*:

```
Thread t = new Thread(() -> expensiveComputation());
```

The general form of a lambda expression is:

```
(A a, B b, C c /* ...possibly more arguments */ ) -> {
  /* Body */
  return /* result */;
}
```

This general form can be lightened in different situations:

- If the compiler can deduce the types of the arguments, which is most commonly the case, you do not have to provide the types (e.g., `(A a, B b)` can be reduced as `(a, b)`).
- If the lambda expression takes one single argument, the parentheses can be removed (e.g., `a -> ...` is a synonym for `(a) -> ...`). Note that a lambda expression with no argument would be defined as `() -> ...`.
- If the body of the lambda expression only contains the `return` instruction, the curly brackets and the `return` can be removed.
- If the lambda expression returns `void` and if its body contains a single line, the curly brackets can be removed as well, for instance:

```
i -> System.out.println(i)
```

- It often happens that you want to write a lambda expression that simply calls a method and passes it the arguments it has received. In such situations, Java offers the notion of **method reference**. For instance, the following lambda expression that calls a static method:

```
i -> System.out.println(i)
```

can be shortened as:

```
System.out::println
```

Similarly, the following lambda expression that calls a non-static method on a list of integers:

```
(List<Integer> a) -> a.size()
```

can be rewritten as:

```
List<Integer>::size
```

6.3 General-purpose functional interfaces

Lambda expressions can only be used in a context that expects a value whose type is a functional interface. It is therefore useful to have a number of such interfaces available, covering the main use cases.

This motivates the introduction of the `java.util.function` standard package that provides general-purpose definitions for:

- Unary functions (with one argument) and binary functions (with two arguments),
- Unary and binary operators (functions whose result type is identical to the type of the argument), and
- Unary and binary predicates (functions whose result type is Boolean).

Make sure to have a look at Java documentation about general-purpose functions: https://docs.oracle.com/javase/8/docs/api/java/util/function/package-summary.html

6.3.1 Unary functions

The `java.util.function.Function` interface represents a **general-purpose function with one argument**. The type `T` of this argument and the result type `R` of the function are the generics parameters of this interface:

```
public interface Function<T,R> {
    public R apply(T t);
}
```

The input type `T` and the result type `R` can be different. Together, they define the **domain** of the function. In mathematical notation, the corresponding function f would be defined as $f : T \mapsto R$.

For instance, the following program first uses a lambda expression to define a function that computes the length of a string, then applies the function to a string:

```
public static void main(String args[]) {
    Function<String, Integer> f = s -> s.length();

    // At this point, no actual computation is done: This is just a definition for "f"!

    System.out.println(f.apply("Hello"));  // Displays: 5
}
```

As another example, here is a function that extracts the first character of a string in lower case:

```
Function<String, Character> f = s -> Character.toLowerCase(s.charAt(0));
System.out.println(f.apply("Hello"));  // Displays: h
```

6.3.2 Binary functions

The `java.util.function.BiFunction` interface represents a **general-purpose functional interface with two arguments** of different types, and with a separate result type:

```
public interface BiFunction<T,U,R> {
    public R apply(T t, U u);
}
```

In mathematical notation, the corresponding function f has domain $f : T \times U \mapsto R$.

In the following example, a lambda expression is used to define a binary function that returns the element of a list at a specific index:

```
BiFunction<List<Float>, Integer, Float> f = (lst, i) -> lst.get(i);

List<Float> lst = Arrays.asList(10.0f, 20.0f, 30.0f, 40.0f);
System.out.println(f.apply(lst, 1));  // Displays: 20
```

6.3.3 Operators

An **operator** is a particular case of a general-purpose functional interface, in which **the arguments and the result are all of the same type**. Operators are so common that Java defines specific interfaces for unary and binary operators:

```
public interface UnaryOperator<T> {
    public T apply(T x);
}

public interface BinaryOperator<T> {
    public T apply(T x, T y);
}
```

The mathematical domain of an unary operator f is $f : T \mapsto T$, whereas the domain of a binary operator f is $f : T \times T \mapsto T$.

As an example, the function computing the square of a double number is an unary operator that could be defined as:

```
UnaryOperator<Double> f = x -> x * x;
System.out.println(f.apply(5.0));  // Displays: 25.0
```

Similarly, for the absolute value:

```
UnaryOperator<Double> f = x -> Math.abs(x);
System.out.println(f.apply(-14.0));    // Displays: 14.0
System.out.println(f.apply(Math.PI));  // Displays: 3.14159...
```

The function computing the sum of two integers can be defined as:

```
BinaryOperator<Integer> f = (x, y) -> x + y;
System.out.println(f.apply(42, -5));  // Displays: 37
```

Remark

If you look at the Java documentation, unary and binary operators are actually defined as:

```
public interface UnaryOperator<T> extends Function<T,T> { }
public interface BinaryOperator<T> extends BiFunction<T,T,T> { }
```

This construction implies that a `UnaryOperator` (resp. `BinaryOperator`) can be used as a placeholder for a `Function` (resp. `BiFunction`). However, the construction is more involved, which explains why we preferred defining the operators as separate interfaces.

6.3.4 Composition

In the context of general-purpose functional interfaces in Java, **composition** refers to the ability to combine multiple functions or operators to create more complex functions. It involves chaining functions together to perform a sequence of operations on data in a concise and expressive manner.

From a mathematical perspective, if we have a function $f : X \mapsto Y$ and a function $g : Y \mapsto Z$, their function composition is the function $g \circ f : X \mapsto Z : x \mapsto g(f(x))$. In other words, the function g is applied to the result of applying the function f to x.

In Java, the `Function` interface contains the default method `compose()` that can be used to construct a new function that represents its composition with another function. Thanks to the fact that `UnaryOperator` is a special case of a `Function`, composition is also compatible with operators.

Here is an example of composition:

```
Function<Integer, Double> f = (i) -> Math.sqrt(i);
UnaryOperator<Double> g = (d) -> d / 2.5;
Function<Integer, Double> h = g.compose(f);

System.out.println(h.apply(25));  // Displays: 2.0, which corresponds to "sqrt(25) / 2.5"
```

Evidently, composition is also available for binary functions and binary operators.

6.3.5 Predicates

Predicates are another particular case of a general-purpose functional interface. They correspond to functions whose **result type is a Boolean value**. Unary predicates are frequently used to filter a collection of objects of a given type. The corresponding functional interface is defined as follows:

```
public interface Predicate<T> {
    public boolean test(T x);
}
```

Pay attention to the fact that while the single abstract method of `Function` is named `call()`, the single abstract method of `Predicate` is named `test()`.

For instance, a predicate that tests whether a list is empty could be defined and used as follows:

```
Predicate<List<Integer>> f = x -> x.isEmpty();

System.out.println(f.test(Arrays.asList()));        // Displays: true
System.out.println(f.test(Arrays.asList(10)));      // Displays: false
System.out.println(f.test(Arrays.asList(10, 20))); // Displays: false
```

Here is another example to test whether a number if negative:

```
Predicate<Double> f = x -> x < 0;

System.out.println(f.test(-10.0)); // Displays: true
System.out.println(f.test(10.0));  // Displays: false
```

Note that there exists a binary version of the `Predicate<T>` unary functional interface, that is known as `BiPredicate<T,U>`.

In the same way functions and operators can be *composed*, the `Predicate` and `BiPredicate` interfaces contain default methods that can be used to create new predicates from existing predicates. Those methods are:

- `and()` to define the logical conjunction of two predicates (i.e., $f \wedge g$),
- `or()` to define the logical disjunction of two predicates (i.e., $f \vee g$), and
- `negate()` to define the logical negation of one predicate (i.e., $\neg f$).

These operations can be used as follows:

```
Predicate<Integer> p = x -> x >= 0;
Predicate<Integer> q = x -> x <= 10;
Predicate<Integer> r = p.and(q);    // x >= 0 && x <= 10
Predicate<Integer> s = p.or(q);     // x >= 0 || x <= 10 <=> true
Predicate<Integer> t = p.negate();  // x < 0

System.out.println(r.test(-5));  // Displays: false
System.out.println(r.test(5));   // Displays: true
System.out.println(r.test(15));  // Displays: false

System.out.println(s.test(-5));  // Displays: true
System.out.println(s.test(5));   // Displays: true
System.out.println(s.test(15));  // Displays: true

System.out.println(t.test(-5));  // Displays: true
System.out.println(t.test(5));   // Displays: false
System.out.println(t.test(15));  // Displays: false
```

6.3.6 Consumer

Finally, a **consumer** is a general-purpose functional interface whose result type is void, i.e., that does not produce any value. It is defined as:

```
public interface Consumer<T> {
    public void accept(T x);
}
```

Consumers are typically encountered as the “terminal block” of a chain of functions. They can notably be used to print the result of a function, to write this result onto a file, or to store this result into another data structure.

For instance, the following code defines a consumer to print the result of a function:

```
Function<Integer, Integer> f = x -> 10 * x;
Consumer<Integer> c = x -> System.out.println(x);

c.accept(f.apply(5));  // Displays: 50
```

6.3.7 Higher-order functions

In Java, **higher-order functions** are methods that can **accept other functions as arguments, return functions as results, or both**. They treat the general-purpose functions seen above as first-class citizens, allowing these functions to be manipulated, passed around, and used as data.

The *composition of two functions* is an example of higher-order function: It takes two `Function` as arguments, and generates one `Function` as its result. We have already seen that Java already provides built-in support for function composition. However, we could have implemented composition by ourselves thanks to the expressiveness of lambda expressions. Indeed, the following program would have produced exactly the same result as the standard `compose()` method of the `Function` class:

```java
public static <X,Y,Z> Function<X,Z> myCompose(Function<Y,Z> g,
                                              Function<X,Y> f) {
    return x -> g.apply(f.apply(x));
}

public static void main(String[] args) {
    UnaryOperator<Double> f = (d) -> d / 2.5;
    Function<Integer, Double> g = (i) -> Math.sqrt(i);
    Function<Integer, Double> h = myCompose(f, g);

    System.out.println(h.apply(25));  // Display: 2.0
}
```

Composition is an example of higher-order function that *outputs* new functions. The standard Java classes also contains methods that take functions as their *inputs*. This is notably the case of the standard Java collections (most notably lists), that include several methods taking operators and predicates as arguments, for instance:

- `forEach(c)` applies a consumer to all the elements of the collection (this is part of the `Iterable<E>` interface),
- `removeIf(p)` removes all of the elements of this collection that satisfy the given predicate (this is part of the `Collection<E>` interface), and
- `replaceAll(f)` replaces each element of the collection with the result of applying the operator to that element (this is specific to the `List<T>` interface).

Here is a full example combining all these three methods:

```java
// Create the following list of integers: [ -3, -2, -1, 0, 1, 2, 3 ]
List<Integer> lst = new ArrayList<>();
for (int i = -3; i <= 3; i++) {
    lst.add(i);
}

// Multiply each integer by 10
lst.replaceAll(x -> 10 * x);  //  => [ -30, -20, -10, 0, 10, 20, 30 ]

// Remove negative integers
lst.removeIf(x -> x < 0);     //  => [ 0, 10, 20, 30 ]

// Print each element in the list
lst.forEach(x -> System.out.println(x));
```

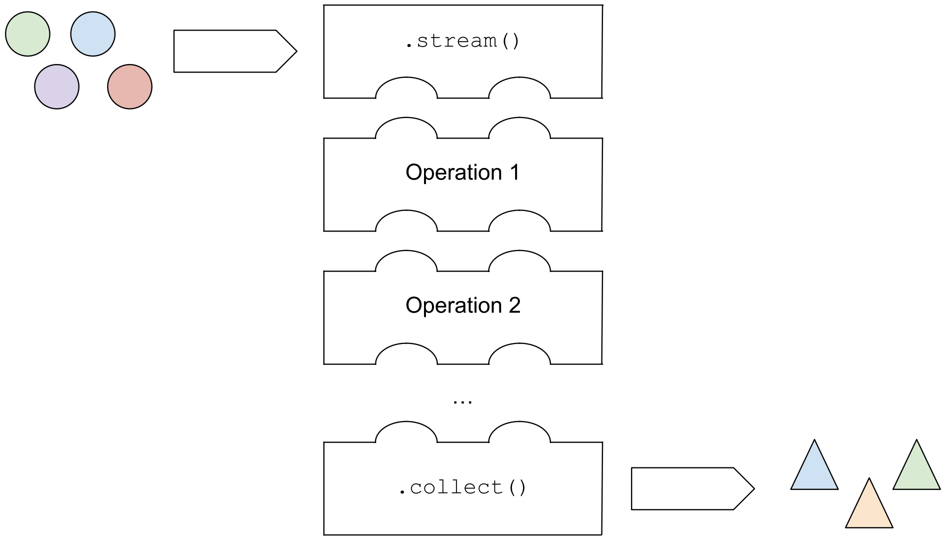
6.4 Streams

In computer science, the term “stream” generally refers to a **sequence of elements accessed one after the other**, from the first to the last.

“Stream programming” in Java refers to the use of the `Stream` API, which was introduced in Java 8 through the `java.util.stream` package. Streams in Java provide a **declarative way to perform computations on sequences of objects**. They enable you to express complex data processing queries more concisely and efficiently than traditional iteration using loops.

Contrarily to Java collections such as `List`, `Set`, or `Map`, streams are *not* a data structure that stores elements in the RAM of the computer. They are a sequence of elements that originate from a data source. This data source can correspond to a Java collection (with elements in RAM), but it might as well correspond to objects that are progressively read from a database, from the filesystem, or from a network communication, possibly without ever being entirely stored inside the RAM of the computer.

In stream programming, the computations to be applied to the individual objects of a stream are declared as a **chain of simple operations** that is called a **stream pipeline**. Lambda expressions are in general used to concisely express these operations. A stream pipeline can be represented as follows:



In this figure:

1. A `Stream` object is first created from a collection of source objects of type `T` (the circles).
2. Zero or more intermediate operations are then successively applied to the individual objects that are part of the input `Stream` object, which generates a new output `Stream` object. These operations can change the content of the input objects, can create new objects (possibly of a different type), or can discard objects.
3. Finally, a terminal operation is applied to collect the results of the stream pipeline. In the figure above, the terminal operation consists in creating an output collection of objects of type `U` (the triangles), which may or may not be the same type as `T`. Other terminal operations are possible, such as counting the number of objects that are produced by the stream pipeline.

6.4.1 Example: From miles to kilometers

To illustrate the benefits of stream programming, let us consider the task of converting a list of strings containing distances expressed in miles, into a list of strings containing the distances expressed in kilometers, ignoring empty strings. For instance, the list containing:

```
[ "15", "", "", "3.5", "" ]
```

should be converted to the list (by definition, 1 mile equals 1609.344 meters):

```
[ "24.14016", "5.6327043" ]
```

Using the Java classes and methods for stream programming, this conversion can be translated very directly into a Java program:

```java
import java.util.List;
import java.util.stream.Collectors;

public class Miles {
    static public void main(String args[]) {
        List<String> miles = List.of("15", "", "", "3.5", "");

        List<String> kilometers = miles.stream()
            .filter(s -> !s.isEmpty())      // Skip empty strings
            .map(s -> Float.parseFloat(s))  // From string to float
            .map(x -> x * 1609.344f)        // From miles to meters
            .map(x -> x / 1000.0f)          // From meters to kilometers
            .map(x -> String.valueOf(x))    // From float to string
            .collect(Collectors.toList());  // Construct the list
    }
}
```

This source code defines a stream pipeline that works as follows:

1. Get a stream from the input list of strings by calling `miles.stream()`.
2. Filter the input stream by removing the empty strings.
3. Parse the strings into floating-point numbers that contain the miles.
4. Convert miles into meters, then convert meters into kilometers.
5. Encode the floating-point numbers that contain the kilometers as strings.
6. Collect all the output strings into a list. This is the terminal operation.

As can be seen, this syntax is very compact and intuitive because it adopts a **declarative** approach. The intermediate operations are expressed using functional-style programming thanks to the *general-purpose functional interfaces* implemented with *lambda expressions*. The intermediate operations are then **chained** together by using the output of one operation as the input of the next.

Importantly, the intermediate operations are not allowed to modify the data they operate on: Each intermediate operation creates a new stream, without modifying its own input stream. This is totally different from the `removeIf()` and `replaceAll()` methods of the *higher-order function of the Java collections*, because the latter methods do modify their data source (i.e., the collection). In other words, stream pipelines have **no side effects** and **do not modify their associated data sources**, as long as the intermediate operations do not modify the state of the program (which is the main assumption of functional programming).

Stream pipelines are also **lazy**, which means that the intermediate operations are not evaluated until a terminal operation is encountered. In the code above, nothing is computed until the `collect()` method is called. This laziness allows for potential optimization by processing only the necessary elements (which is for instance useful if the data source corresponds to a large file), and by opening the path to the exploitation of multiple threads to separately process successive elements in a stream.

6.4.2 Java streams

Stream programming in Java is built on the top of the following generic interface:

```java
package java.util.stream;

public interface Stream<T> {
    // Member methods
}
```

This interface represents a stream of elements of type `T`. In our *previous example*, the streams of string values are of type `Stream<String>`, whereas the streams of floating-point values are of type `Stream<Float>`. We could have made this more explicit by splitting the chain of the intermediate operations as distinct streams, which would have led to the equivalent (but less elegant) code:

```java
import java.util.List;
import java.util.stream.Collectors;
import java.util.stream.Stream;

public class Miles {
    static public void main(String args[]) {
        List<String> miles = List.of("15", "", "3.5", "");

        Stream<String> s1 = miles.stream();
        Stream<String> s2 = s1.filter(l -> !l.isEmpty());
        Stream<Float>  s3 = s2.map(Float::parseFloat);
        Stream<Float>  s4 = s3.map(f -> f * 1609.344f);
        Stream<Float>  s5 = s4.map(f -> f / 1000.0f);
        Stream<String> s6 = s5.map(String::valueOf);

        List<String> kilometers = s6.collect(Collectors.toList());
    }
}
```

Methods that work on streams belong to exactly one of the following three categories:

- **Source methods**, which produce a stream of elements from a source (such as a collection, a file, a database,…),
- **Intermediate methods**, which transform the elements from an input stream to produce a new stream, and
- **Terminal methods**, which consume the elements in the stream (for instance, to print them on screen, to write them into a file, to store them in a Java collection, or to extract a single value out of them).

In the miles-to-kilometers conversion, the `stream()` method is the source method, the `filter()` and `map()` methods are the intermediate methods, and the `collect()` method is the terminal method. Altogether, these methods define the stream pipeline.

Note that streams are not immutable, in the sense that when the elements of a stream are consumed by a terminal method, they disappear and the stream is then empty. In this respect, streams are similar to *iterators*, which are also modified as they are traversed.

Important remark

Note that **you can only go once through the same stream pipeline**. Once an element has been consumed, it is not possible anymore to access the same element again. For instance, this code will *not* work:

```
Stream<Integer> stream = List.of(1, 2, 3, 4, 5).stream();
List<Integer> a = stream.collect(Collectors.toList());  // OK
List<Integer> b = stream.collect(Collectors.toList());  // => java.lang.
↪IllegalStateException: stream has already been operated upon or closed
```

The main source, intermediate, and terminal methods for stream programming in the Java standard library are now reviewed. Evidently, this list is by no way exhaustive. The full list of the features offered `Stream<T>` is available in the online Java documentation: https://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html

6.4.3 Source methods

As already shown in the *miles-to-kilometers example*, **streams are frequently created out of one of the standard Java collections** (such as `List` or `Set`). This stems from the fact that the `Collection<E>` interface contains the following method:

```
public interface Collection<E> {
    // Other members

    default Stream<E> stream();
}
```

This source method provides a bridge between the world of Java collections and the world of Java streams. As already outlined before, the resulting stream uses the collection as its data source, but it never modifies this data source.

It is also possible to directly create a stream without using a collection. For instance, the `Stream<T> empty()` static method creates an **empty stream**:

```
Stream<String> s = Stream.empty();
```

A stream containing a **predefined list of values** can be constructed using the `Stream<T> of(T... values)` static method. For instance, here is how to define a stream containing the vowels in French:

```
Stream<Character> vowels = Stream.of('a', 'e', 'i', 'o', 'u', 'y');
```

Note that the type `T` of the `of()` method can be a custom class. For instance, the following code is perfectly valid and creates a stream of complex objects whose class is `Account`:

```
class Account {
    private String name;
    private int value;

    public Account(String name,
                   int value) {
        this.name = name;
        this.value = value;
    }

```

(continues on next page)

(continued from previous page)

```
    public Account(int value) {
        this.name = "";
        this.value = value;
    }

    public String getName() {
        return name;
    }

    public int getValue() {
        return value;
    }
}

Stream<Account> accounts = Stream.of(new Account(100), new Account(200));
```

Streams can also be **constructed out of arrays** using the `Arrays.stream()` static method. For instance:

```
Float[] a = new Float[] { 1.0f, 2.0f, 3.0f };
Stream<Float> b = Arrays.stream(a);
```

As far as files are concerned, the method `lines()` of the standard `BufferedReader` class returns a stream that scans the lines of a *Java reader*. This notably enables the creation of a **stream that reads the lines of a file**:

```
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
import java.util.stream.Stream;

public class FileToStream {
    static public void main(String args[]) throws IOException {
        try (FileReader file = new FileReader("somefile.txt")) {
            BufferedReader reader = new BufferedReader(file);
            Stream<String> lines = reader.lines();
        }
    }
}
```

Finally, because streams are lazy, it is possible to define **streams that span an infinite number of elements**. Obviously, an infinite sequence cannot be entirely computed before being accessed. The trick is to provide a seed element from which the sequence begins, together with an *unary operator* that continuously updates the seed element to generate the next element in the sequence. The static method `iterate()` of the `Stream<T>` interface is introduced to this end:

```
static <T> Stream<T>  iterate(T seed, UnaryOperator<T> f);
```

The stream created by `iterate()` will successively produce the elements `seed`, `f(seed)`, `f(f(seed))`, `f(f(f(seed)))`... These successive elements are only computed on demand, i.e., when the next stream in the stream pipeline has to read a new element from its input stream. For instance, here is how to define the infinite stream of even positive numbers:

```
Stream<Integer> evenNumbers = Stream.iterate(0, x -> x + 2);
```

Infinite streams can also be created using the static method `generate()` of the `Stream<T>` interface:

```
public interface Supplier<T> {
    public T get();
}

static <T> Stream<T>  generate(Supplier<T> s);
```

The `generate()` factory method is more generic than the `iterate()` factory method, in the sense that `generate()` allows to manage a context that is richer than a single seed element using the members of the class implementing the `Supplier<T>` interface. Note that the `Supplier<T>` functional interface looks very similar to `Callable<T>`, except that its single abstract method is named `get()` instead of `call()`. The example below illustrates how to create a supplier that generates a sequence of integers, starting from a seed value and successively adding a delta value, and how to use it to generate the sequence of negative odd numbers:

```
class IntegerSequenceSupplier implements Supplier<Integer> {
    private int value;
    private int delta;

    IntegerSequenceSupplier(int seed,
                            int delta) {
        this.value = seed;
        this.delta = delta;
    }

    @Override
    public Integer get() {
        int current = value;
        value = value + delta;
        return current;
    }
}

Stream<Integer> oddNumbers = Stream.generate(new IntegerSequenceSupplier(-1, -2));
```

Evidently, the same stream could have been generated as:

```
Stream.iterate(-1, x -> x - 2);
```

6.4.4 Intermediate methods

Once an instance of `Stream<T>` is created using one of the *source methods*, various intermediate operations can be applied to it.

Map

The method `map(Function<T,R> f)` is especially important and very frequently used, as it allows you to **transform the elements of a stream** to obtain a new one. The `map()` method is a *higher-order function* that takes as argument an *unary function* `f` that is generally expressed as a *lambda expression*. This unary function `f` is applied to the elements of the stream, which leads to the creation of an output stream. As an example, here is how to increment each value in a stream of integers:

```
Stream<Integer> stream = Stream.of(10, 20, 30, 40, 50);
Stream<Integer> incremented = stream.map(i -> i + 1);  // => [11, 21, 31, 41, 51]
```

This is an example that uses an *unary operator*, as both streams share the same data type (i.e., `Integer`). But the `map()` method also accept general functions that change the data type of the elements of the input stream. As an example, the following code creates a stream that provides the number of characters in each element of a stream of strings:

```
Stream<String> stream = Stream.of("Bonjour", "Hello");
Stream<Integer> lengths = stream.map(s -> s.length());  // This unary function goes from String to Integer
// Note that we could have written: stream.map(String::length);
```

The `map()` method evidently supports user-defined classes. For instance, it is possible to extract the values from a stream of `Account` objects (as introduced in the *previous section*) as a stream of integers:

```
Stream<Account> accounts = Stream.of(new Account(100), new Account(200));
Stream<Integer> values = accounts.map(i -> i.getValue());  // => [100, 200]
// Note that we could have written: accounts.map(Account::getValue);
```

Interestingly, the `map()` method can also be used to create objects by calling their constructor on the elements from the input stream. For instance:

```
Stream<Integer> values = Stream.of(100, 300, 600);
Stream<Account> accounts = values.map(x -> new Account(x));
// Note that we could have written: values.map(Account::new);
```

Filter

Another significant method on streams is `filter(Predicate<T> p)`. This method is a higher-order function that takes as a argument an *unary predicate* `p`, and that creates a new stream that **only contains the objects from the input stream that verify the predicate**. As an example, here is how to filter a stream of integers to keep only the values that are integer multiples of 4:

```
Stream<Integer> stream = Stream.of(10, 20, 30, 40, 50);
Stream<Integer> filtered = stream.filter(i -> i % 4 == 0);  // [20, 40]
```

Other intermediate methods

The `map()` and `filter()` methods are extremely important higher-order functions that also exist in Python to manipulate lists. Besides `map()` and `filter()`, let us also highlight the existence of the following intermediate methods:

- `sorted()` returns a stream containing the same elements as the one to which it is applied, but **sorted in ascending natural order**:

```
Stream<Integer> stream = Stream.of(20, 10, 50, 40, 30);
Stream<Integer> sorted = stream.sorted();  // => [10, 20, 30, 40, 50]
```

- `sorted(Comparator<T> c)` returns a stream containing the same elements as the one to which it is applied, but **sorted by delegation to a comparator**:

```
Stream<Account> accounts = Stream.of(new Account(500), new Account(100), new␣
↪Account(800), new Account(200));
Stream<Account> sorted = accounts.sorted((a, b) -> a.getValue() - b.getValue());
// => [ Account(100), Account(200), Account(500), Account(800) ]
```

- `skip(long n)` **ignores the first “n” elements** from the input stream, and returns the stream of the remaining elements:

```
Stream<Integer> stream = Stream.of(10, 20, 30, 40, 50);
Stream<Integer> skipped = stream.skip(3);  // => [40, 50]
```

- `limit(long l)` returns a stream containing the same elements as the stream to which it is applied, but **truncated to have at most “l” elements**:

```
Stream<Integer> stream = Stream.of(10, 20, 30, 40, 50);
Stream<Integer> limited = stream.limit(2);  // => [10, 20]
```

  The `limit()` method can notably be used to truncate infinite streams into a finite stream.

6.4.5 Terminal methods

In a stream pipeline, the last operation is usually an operation that returns some result that is not a stream: This last operation **gets the data out of the stream**. It is possible to distinguish between three different cases:

- Stream pipelines whose terminal method does something with each individual element of the stream (**consumer methods**),
- Stream pipelines whose terminal method creates a Java data structure to store the elements of the stream (**collector methods**), and
- Stream pipelines whose terminal method extracts a single value out of the stream (**reduction methods**).

Consumer methods

The `forEach()` terminal method of the `Stream<T>` interface is a higher-order function that takes as input a *consumer function* `c`. This method applies the `c` function to all the elements of the stream. This is similar to the `map()` *intermediate method*, but since consumers do not produce an output value, the `forEach()` operation yields no result.

Typical usages of `forEach()` include printing the content of the stream, writing it to a file, sending it over a network connection, or saving it to a database. A very common pattern consists in calling the standard `System.out.println()` method on each element in the stream:

```
Stream<String> stream = Stream.of("Hello", "World");
stream.forEach(s -> System.out.println(s));
```

By virtue of the *method reference* construction, the code above is often shortened as:

```
Stream<String> stream = Stream.of("Hello", "World");
stream.forEach(System.out::println);
```

Collector methods

Another possibility for a terminal method consists in collecting the elements of the stream into a Java data structure. This is the role of the `collect()` method that is available in the `Stream<T>` interface, and takes as argument an object that implements the `Collector` interface.

The `Collector` interface represents a very generic construction that is quite complex to master. Fortunately, Java proposes a set of predefined, concrete implementations of the `Collector` interface that can be directly instantiated using the static methods of the `java.util.stream.Collectors` class: https://docs.oracle.com/javase/8/docs/api/java/util/stream/Collectors.html

In what follows, the return type of the static methods of `Collectors` is never shown, as it tends to be complicated and understanding it is not necessary for the use of the predefined collectors. In practice, the result of the static methods is always passed directly to the `collect()` method of a stream.

Here are some of the most useful predefined collectors to create Java containers from streams:

- `Collectors.toList()` returns a collector that **stores the elements of the stream into a list**. If the stream is of type `Stream<T>`, the output list will be of type `List<T>`. For instance:

  ```
  Stream<Integer> stream = Stream.of(10, 20, 30);
  List<Integer> lst = stream.collect(Collectors.toList());  // => List: [10, 20, 30]
  ```

- `Collectors.toSet()` returns a collector that **stores the elements of the stream into a set**. If the stream is of type `Stream<T>`, the output list will be of type `Set<T>`. For instance:

  ```
  Stream<Integer> stream = Stream.of(10, 20, 20, 10);
  Set<Integer> lst = stream.collect(Collectors.toSet());  // => Set: {10, 20}
  ```

- `Collectors.toMap(Function<T,K> keyMapper, Function<T,U> valueMapper)` returns a collector that **stores the elements of the stream into an associative array** (i.e., a dictionary). The function `keyMapper` is used to generate the key corresponding to each element in the stream, and the function `valueMapper` is used to generate the value corresponding to each element. If the stream is of type `Stream<T>`, the output list will be of type `Map<K,V>`. For instance:

  ```
  Stream<Account> accounts = Stream.of(new Account("Dupont", 100),
                                       new Account("Dupond", 200));
  ```

(continues on next page)

(continued from previous page)

```
Map<String, Integer> nameToValue =
    accounts.collect(Collectors.toMap(x -> x.getName(),
                                      x -> x.getValue()));
// The associative array will contain: { "Dupont" : 100, "Dupond" : 200 }
```

If the stream is of type Stream<String>, the following predefined containers can be used to combine the successive strings from the stream:

- Collectors.joining() returns a collector that **concatenates the individual strings from a stream of strings**. For instance:

```
Stream<String> stream = Stream.of("one", "two", "three");
String joined = stream.collect(Collectors.joining());
System.out.println(joined);  // Displays: onetwothree
```

- Collectors.joining(String delimiter) works similarly to Collectors.joining(), but it adds the delimiter between each individual string:

```
Stream<String> stream = Stream.of("one", "two", "three");
String joined = stream.collect(Collectors.joining(", "));
System.out.println(joined);  // Displays: one, two, three
```

- Collectors.joining(String delimiter, String prefix, String suffix) works similarly to Collectors.joining(delimiter), but it also add a prefix and a suffix:

```
Stream<String> stream = Stream.of("one", "two", "three");
String joined = stream.collect(Collectors.joining(", ", "{ ", " }"));
System.out.println(joined);  // Displays: {one, two, three}
```

Finally, note that the Stream<T> interface also contains the toArray() collector method. This method create an array of Object from a stream:

```
Stream<Integer> stream = Stream.of(100, 200);
Object[] a = stream.toArray();
System.out.println((Integer) a[0]);  // Displays: 100
System.out.println((Integer) a[1]);  // Displays: 200
```

The downside of toArray() is that the generic type T is lost and replaced by type Object, i.e., the root of the class hierarchy. This opens a huge risk of *bad casts*. It is therefore generally better to use the predefined collectors instead of toArray().

Reduction methods

As explained above, consumer methods apply an operation to each element in a stream, while collector methods create a new Java data structure that combines all the elements of a stream. This contrasts with reduction methods, that produce **one single value from the entire stream**.

The simplest reduction method consists in **counting the number of elements** in the stream. The count() method can be used to this end:

```
Stream<String> stream = Stream.of("one", "two", "three");
System.out.println(stream.count());  // Displays: 3
```

Reduction methods also exist to test to which extent the elements of a stream **satisfy a common logical condition** expressed as a *predicate*:

- `allMatch(Predicate<T> p)` returns `true` if and only if all the individual elements in a `Stream<T>` satisfy the predicate `p`.
- `anyMatch(Predicate<T> p)` returns `true` if and only if at least one of individual elements in a `Stream<T>` satisfies the predicate `p`.
- `noneMatch(Predicate<T> p)` returns `true` if and only if none of the individual elements in a `Stream<T>` satisfies the predicate `p`.

For instance, the following code tests whether we are given a stream of even integers:

```
System.out.println(Stream.of(2, 4, 8, 12).allMatch(x -> x % 2 == 0));  // Displays: true
System.out.println(Stream.of(2, 4, 8, 13).allMatch(x -> x % 2 == 0));  // Displays: false
```

The most generic reduction method is offered by the `reduce()` method. There exists several variants of this method, but the most commonly used is `T reduce(T identity, BinaryOperator<T> accumulator)`: This method returns the value resulting from **successive application of the binary operator** `accumulator` to the initial value `identity` and to the successive elements of the stream. If the stream is empty, this method simply returns `identity`.

As an example, let us consider the task of computing the **sum of the values in a stream of integers**. Zero being the identity element for the addition, the `reduce()` method could be used as follows:

```
Stream<Integer> stream = Stream.of(2, 5, 8);
System.out.println(stream.reduce(0, (a, b) -> a + b));  // Displays: 15
```

Here are the steps of the computation that is carried on by the `reduce()` method:

1. It initializes a temporary variable whose initial value is given by `identity` (i.e., `0` in this case).
2. It reads the next element from the stream, for instance `2`. Using the lambda expression for `accumulator`, it computes `0 + 2` (where `0` is the temporary variable), and stores the result `2` in the temporary variable.
3. It reads the next element, for instance `5`. According to `accumulator`, it computes `2 + 5` (where `2` is the temporary variable), and stores `7` in the temporary variable.
4. It reads the final element, for instance `8`. According to `accumulator`, it computes `7 + 8` (where `7` is the temporary variable), and stores `15` in the temporary variable, which is at last returned to the caller.

As another example, here is how to compute the **product of the values in a stream of double precision numbers**, for which `1.0` is the identity value:

```
Stream<Double> stream = Stream.of(-1.0, 2.0, 5.5);
System.out.println(stream.reduce(1.0, (a, b) -> a * b));  // Displays: -11.0
```

Remark

Pay attention to the fact that during a reduction, the order in which the binary operator `accumulator` is applied to the various elements is not specified (for instance because parallelism can possibly be used to speed up some computation), so this operator must be associative.

6.4.6 About laziness

Let us consider the following code:

```
Stream<Integer> s1 = Stream.of(1, 2, 3, 4, 5);
Stream<Integer> s2 = s1.map(i -> { System.out.println(i); return i + 1; });
```

You could think that the code does the following:

1. A stream with elements 1, 2, 3, 4, and 5 is created.
2. The lambda expression `i -> { System.out.println(i); return i + 1; }` is applied to each element.
3. The console display the lines 1, 2, 3, 4, and 5.
4. A new stream containing 2, 3, 4, 5, and 6 is returned.

However, this is wrong! The code above does not print anything. Indeed, as *written above*, **streams are lazy**. The operations are only executed if the result is needed, for example as in:

```
Stream<Integer> s1 = Stream.of(1, 2, 3, 4, 5);
Stream<Integer> s2 = s1.map(i -> { System.out.println(i); return i + 1; });
Object[] a = s2.toArray();  // <- here, all the elements of the stream are needed
```

This code will actually print all the elements onto the console! Now, let us consider the following example:

```
Stream<Integer> s1 = Stream.of(1, 2, 3, 4, 5);
Stream<Integer> s2 = s1.map(i -> { System.out.println(i); return i + 1; });
Object[] a = s2.limit(2).toArray();  // <- here, only the two first elements of the stream are needed
```

This code will only print `1` and `2` onto the console! This is because the `limit()` method stops further processing of its input stream as soon as the maximum number of elements is reached. This also explains why *infinite streams* can be represented using Java streams.

It is also interesting to understand that the progression over interdependent streams is **interleaved**. This can be seen in the following example:

```
Stream<Integer> s1 = Stream.of(1, 2, 3);
Stream<Integer> s2 = s1.map(i -> { System.out.println("a: " + i); return i + 1; });
s2.forEach(j -> System.out.println("b: " + j));
```

This example prints the following sequence:

```
a: 1
b: 2
a: 2
b: 3
a: 3
b: 4
```

This is because the reading of streams `s1` and `s2` is interleaved. Here are the steps of the evaluation:

1. `forEach()` needs the first element of `s2`. To obtain this first element, `map()` is executed on the first element of `s1`. So, `a: 1` is printed, then `b: 2`.
2. `forEach()` needs the second element of `s2`. To obtain this second element, `map()` is executed on the second element of `s1`. So, `a: 2` is printed, then `b: 3`.

3. forEach() needs the third and last element of s3. To obtain this last element, map() is executed on the last element of s1. So, a: 3 is printed, then b: 4.

Finally, because streams are lazy, they can be also used in situations **where it is not known in advance how long the stream is**. For instance, here is a sample code that prints all the lines of a text file in upper case, by using the lines() *source method of a reader*:

```
try (FileReader file = new FileReader("somefile.txt")) {
    BufferedReader reader = new BufferedReader(file);
    Stream<String> lines = reader.lines();
    lines.map(String::toUpperCase).forEach(System.out::println);
}
```

This code does *not* read the entire file, then print it. That would use a lot of memory if the file is very big! Instead, this code reads a new line from the file *only when it is needed*. In other words, the code reads the first line, prints it, reads the second line, prints it, and so on.

6.4.7 Specialized streams

So far, we have only been considering the generic interface Stream<T>. For performance reasons, there also exist **specialized interfaces representing streams of three primitive types**, namely:

- IntStream for streams of int numbers: https://docs.oracle.com/javase/8/docs/api/java/util/stream/IntStream.html
- LongStream for streams of long numbers: https://docs.oracle.com/javase/8/docs/api/java/util/stream/LongStream.html
- DoubleStream for streams of double precision numbers (i.e., for the primitive type double): https://docs.oracle.com/javase/8/docs/api/java/util/stream/DoubleStream.html

For instance, a stream of int numbers can either be represented as a generic stream of type Stream<Integer> (in which each element is an object of type Integer), or as a specialized stream of type IntStream (in which each element is internally represented as an int primitive value). Specialized streams are in general more efficient than generic streams, as they avoid the creation of objects, and should be preferred wherever possible when performance optimization or memory usage is important.

Note that there is no specialized streams for the other primitive types: It is recommended to use IntStream to store short, char, byte, and boolean values. As far as float are concerned, it is recommended to use DoubleStream if a specialized stream is preferable.

The **conversions between generic streams and specialized streams** are ruled as follows:

- A specialized IntStream can be constructed from a Stream<T> using the mapToInt() method of the generic stream.
- A specialized LongStream can be constructed from a Stream<T> using the mapToLong() method of the generic stream.
- A specialized DoubleStream can be constructed from a Stream<T> using the mapToDouble() method of the generic stream.
- Conversely, a generic Stream<T> can be constructed from an IntStream, from a LongStream, or from an DoubleStream using the mapToObj() method of the specialized stream.

For instance, the *conversion between miles and kilometers* could have been implemented as follows using the specialized DoubleStream (the two modified lines are highlighted by asterisks):

```
List<String> kilometers = miles.stream()
    .filter(s -> !s.isEmpty())                // Skip empty strings
    .mapToDouble(s -> Double.parseDouble(s))  // (*) From string to double
    .map(x -> x * 1609.344)                   // From miles to meters
    .map(x -> x / 1000.0)                     // From meters to kilometers
    .mapToObj(x -> String.valueOf(x))         // (*) From double to string
    .collect(Collectors.toList());            // Construct the list
```

As another example, here is how to create an `IntStream` from the elements of a stream of `Account` objects (as introduced in the *previous section*):

```
Stream<Account> accounts = Stream.of(new Account(100), new Account(200));
IntStream values = accounts.mapToInt(i -> i.getValue());
values.forEach(System.out::println);
```

It is also worth noticing that `IntStream` and `LongStream` have two interesting static methods `range()` and `rangeClosed()` that can be used to easily create a **stream that represents an interval of integers**:

```
IntStream s1 = IntStream.range(-3, 3);         // => [-3, -2, -1, 0, 1, 2]
IntStream s2 = IntStream.rangeClosed(-3, 3);   // => [-3, -2, -1, 0, 1, 2, 3]
```

As can be seen in this example, these two methods only differ with respect to the fact that the last integer in the range is included or not.

Finally, besides their interest for optimization, the specialized streams also provide convenient *collector methods* that **directly return Java arrays from a stream**:

- The `toArray()` method of `IntStream` creates a `int[]` value,
- The `toArray()` method of `LongStream` creates a `long[]` value, and
- The `toArray()` method of `DoubleStream` creates a `double[]` value.

Here is an example of this feature:

```
IntStream stream = IntStream.of(10, 20, 30);
int[] a = stream.toArray();
```

6.5 Programming without side effects

6.5.1 Side effects

Remember that *lambda expressions* in Java are implemented as anonymous inner classes. As a consequence, they are allowed to access members of the outer class:

```
import java.util.function.Function;

public class SideEffect {
    private int sum = 0;

    public Function<Integer, Integer> createAddition() {
        return i -> {
            sum++;           // Side effect!
```

(continues on next page)

(continued from previous page)

```
            return i + sum;
        };
    }

    public static void main(String[] args) {
        Function<Integer,Integer> add = new SideEffect().createAddition();

        System.out.println(add.apply(3));  // Displays: 4
        System.out.println(add.apply(3));  // Displays: 5
    }
}
```

This is an extremely counter-intuitive behavior: Any developer that has not implemented the `createAddition()` method would expect that multiple applications of the same `Function` should always give the same result. In mathematics, we indeed expect that a function always gives the same result for the same argument!

For this reason, this kind of code should be avoided, even if it fully respects the Java syntax. **A good function should have no side effect**. A function should never change existing objects and variables. The code of a function or method is easier to understand if the result *only* depends on its arguments. Furthermore, a function that has side effect can result in severe concurrency issues if executed in a multithreaded context.

6.5.2 Immutable objects

In order to enforce the absence of side effects and to ensure thread safety, functional programming promotes the use of **immutable objects**. Immutable objects are objects whose state cannot be changed after they are created.

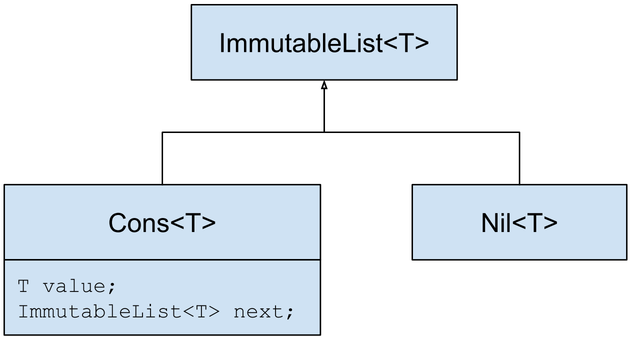
For instance, **strings in Java are immutable**. Once a `String` object is created, its value cannot be changed. Operations that appear to modify a string actually create a new `String` object.

On the other hand, primitive types in Java *are* mutable (for instance, you can change the value of a `int` variable after its declaration). However, the **wrapper classes associated with primitive types are immutable**. Indeed, classes like `Integer`, `Long`, `Float`, `Double`, `Byte`, `Short`, `Character`, or `Boolean`, which are used to wrap the primitive data types, are all immutable: The primitive value they store can never be changed after their construction. In the same vein, *Java enums* are implicitly immutable. Once the enum constants are created, their values cannot be modified.

6.5.3 An immutable list

We already know that the standard `List<T>` interface offers the `removeIf()` and `replaceAll()` methods to apply *higher-order functions* onto their content. Because of the presence such methods, the standard Java lists are mutable objects, which contrasts with the philosophy of functional programming. Can we design a list without side effect, in a way that is similar to streams? The answer is “yes”, and this section explains how to **create an immutable generic list**.

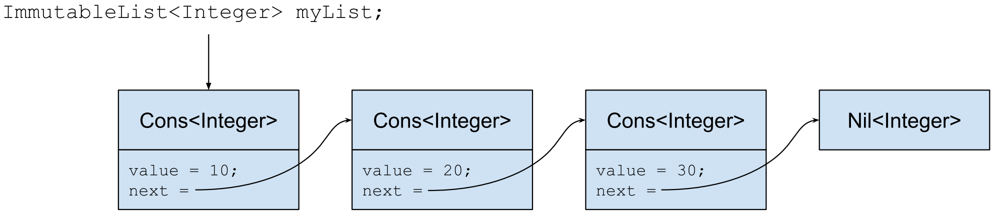
The basic idea is similar to the *linked list abstract data structure*. It consists in creating the following class hierarchy:



In this hierarchy:

- `T` is a generic type that must correspond to an immutable type, such as `Integer` or `Float`.
- `Nil<T>` represents the end of an immutable list. This data structure contains no information, it is simply a terminal node.
- `Cons<T>` stores a non-empty value of the list, together with a link to the next element in the list. The name `Cons` originates from the fundamental `cons` function that is used in most dialects of the Lisp programming language to construct memory objects.
- `ImmutableList<T>` is an *interface* that represents a reference to the immutable list itself.

This class hierarchy would store the immutable list containing the 10, 20, and 30 integer values as follows:



Here is the corresponding Java code to define this hierarchy:

```java
public interface ImmutableList<T> {
}

public static final class Nil<T> implements ImmutableList<T> {
    public Nil() {
    }
}

public static final class Cons<T> implements ImmutableList<T> {
    private final T value;
    private final ImmutableList<T> next;

    public Cons(T value,
                ImmutableList<T> next) {
        if (next == null) {
            throw new IllegalArgumentException();
        }
        this.value = value;
        this.next = next;
    }
}
```

And our sample immutable list containing 10, 20, and 30 can then be constructed from its last element to its first element:

```java
// Create the terminal node that represent the empty list
ImmutableList<Integer> empty = new Nil<Integer>();

// Create the node containing "30" that represents the list: [ 30 ]
ImmutableList<Integer> list1 = new Cons<Integer>(30, empty);

// Create the node containing "20" that represents the list: [ 20, 30 ]
ImmutableList<Integer> list2 = new Cons<Integer>(20, list1);

// Create the node containing "10" that represents the list: [ 10, 20, 30 ]
ImmutableList<Integer> myList = new Cons<Integer>(10, list2);
```

Note that `list1` and `list2` are always the same lists: We can add an element to the head of a list without changing the tail of the list. This demonstrates that **this data structure is immutable**. It cannot be changed after creation.

6.5.4 Consuming an immutable list

Because our `ImmutableList<T>` data structure is immutable, we do not provide a public access to the values it stores.

If we want to execute an operation on each element of the immutable list according to the functional programming paradigm, we would use a *consumer*. A consumer can be applied to our immutable list by adding a `forEach()` higher-order method in the `ImmutableList<T>` interface, which mimics *streams*:

```java
public interface ImmutableList<T> {
    public void forEach(Consumer<T> consumer);
}
```

The implementation of `forEach()` in the concrete classes `Nil<T>` and `Cons<T>` follows a *recursive algorithm*. The base case of the recursion corresponds to the handling of an empty list, which simply does nothing:

```java
public static final class Nil<T> implements ImmutableList<T> {
    // ...

    public void forEach(Consumer<T> consumer) {
    }
}
```

As far as a non-empty node is concerned, the concrete class first applies the consumer to the value that it stores, then it forwards the consumer to the next item in the list:

```java
public static final class Cons<T> implements ImmutableList<T> {
    private final T value;
    private final ImmutableList<T> next;
    // ...

    public void forEach(Consumer<T> consumer) {
        consumer.accept(value);
        next.forEach(consumer);
    }
}
```

Thanks to this `forEach()` method, it becomes possible to print the elements of the immutable list:

```
myList.forEach(i -> System.out.println(i));
```

6.5.5 Map and filter on an immutable list

Since an immutable list cannot be changed, we have to create a new immutable list if we want to change the content of the list. To this end, let us implement the `map()` and `filter()` methods that are *the most common intermediate operations in a stream pipeline*. We first add the two abstract methods to our `ImmutableList<T>` interface:

```
public interface ImmutableList<T> {
    public void forEach(Consumer<T> consumer);

    public ImmutableList<T> map(UnaryOperator<T> operator);

    public ImmutableList<T> filter(Predicate<T> predicate);
}
```

In the base case of an empty immutable list, both `map()` and `filter()` simply have to return a new empty list:

```
public static final class Nil<T> implements ImmutableList<T> {
    // ...

    public ImmutableList<T> map(UnaryOperator<T> operator) {
        return new Nil<T>();
    }

    public ImmutableList<T> filter(Predicate<T> predicate) {
        return new Nil<T>();
    }
}
```

As far as a non-empty node is concerned, the concrete class is implemented as follows:

```
public static final class Cons<T> implements ImmutableList<T> {
    private final T value;
    private final ImmutableList<T> next;
    // ...

    public ImmutableList<T> map(UnaryOperator<T> operator) {
        // Create a new list with the transformed current value, followed by the␣
↪transformed tail of the list
        return new Cons<T>(operator.apply(value), next.map(operator));
    }

    public ImmutableList<T> filter(Predicate<T> predicate) {
        if (predicate.test(value)) {
            return new Cons<T>(value, next.filter(predicate));
        } else {
            return next.filter(predicate);
        }
    }
}
```

Let us for instance consider the operation `myList.map(x -> x + 3)` on our immutable list containing 10, 20, and 30. If we expand the recursive calls, we get the following sequence of operations:

```
myList.map(x -> x + 3)
<=> new Cons(10 + 3, list2.map(x -> x + 3))
<=> new Cons(10 + 3, new Cons(20 + 3, list3.map(x -> x + 3)))
<=> new Cons(10 + 3, new Cons(20 + 3, new Cons(30 + 3, empty.map(x -> x + 3))))
<=> new Cons(10 + 3, new Cons(20 + 3, new Cons(30 + 3, new Nil())))
<=> ImmutableList<Integer> containing: [13, 23, 33]
```

Thanks to those `map()` and `filter()` methods, we can chain operations to create more complex pipelines, for instance:

```java
myList.filter(x -> x > 2).map(x -> 2 * x).forEach(System.out::println);
```

Evidently, our `ImmutableList<T>` data structure could be further improved by replicating all the methods that are available to design *stream pipelines*.

CHAPTER SEVEN

INDICES AND TABLES

- genindex
- modindex
- search
- `PDF version of the book`

BIBLIOGRAPHY

[HS65] Juris Hartmanis and Richard E Stearns. On the computational complexity of algorithms. *Transactions of the American Mathematical Society*, 117:285–306, 1965.